

A PROTOTYPE RING INTERFACE
FOR THE NPS DATA COMMUNICATION RING

Michael James Harris

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93940

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

A PROTOTYPE RING INTERFACE
FOR THE NPS DATA COMMUNICATION RING

by

Michael James Harris

June 1974

Thesis Advisor:

R. H. Brubaker, Jr.

Approved for public release; distribution unlimited.

T 16 1508

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Prototype Ring Interface For The NPS Data Communication Ring		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1974
7. AUTHOR(s) Michael James Harris		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1974
		13. NUMBER OF PAGES 110
		15. SECURITY CLASS. (of this report) Unclassified
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Ring Interface Data Communication Ring Ring Network Ring Protocol Ring Repeater Distributed Computer System		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The architecture and hardware/firmware design of a prototype microcontrolled Ring Interface (RI) for the proposed Naval Postgraduate School Data Communication Ring System is presented. The theory and conceptual design of the Data Communication Ring System is based upon the thesis research of Lieutenant Keith Albert Hirt (December 1973) and is the basis for the protocol and data processing algorithms used in the hardware design of the Ring Interface. A microcontroller and its associated assembly language,		

20. (cont'd)

SMAL (Symbolic Microcontroller Assembly Language), are discussed and the Ring Interface Program (written in SMAL) is also included along with the respective machine code version. Finally, the expected capabilities, limitations, and tradeoffs are presented with possible long-term improvements.

A Prototype Ring Interface
For the NPS Data Communication Ring

by

Michael James Harris
Ensign, United States Navy
B.S., United States Naval Academy, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1974

Ther's
2/5/2007
C.

ABSTRACT

The architecture and hardware/firmware design of a prototype microcontrolled Ring Interface (RI) for the proposed Naval Postgraduate School Data Communication Ring System is presented. The theory and conceptual design of the Data Communication Ring System is based upon the thesis research of Lieutenant Keith Albert Hirt (December 1973) and is the basis for the protocol and data processing algorithms used in the hardware design of the Ring Interface. A microcontroller and its associated assembly language, SMAL (Symbolic Microcontroller Assembly Language), are discussed and the Ring Interface Program (written in SMAL) is also included along with the respective machine code version. Finally, the expected capabilities, limitations, and tradeoffs are presented with possible long-term improvements.

TABLE OF CONTENTS

Form DD 1473.....	1
LIST OF FIGURES.....	6
ACKNOWLEDGMENTS.....	7
I. INTRODUCTION.....	8
II. DATA HANDLING TECHNIQUES.....	12
A. TRANSMISSION.....	12
B. RECEPTION.....	14
III. CONTROL PROTOCOL.....	16
A. TOKENS.....	16
B. TIMING HIERARCHY.....	20
IV. RING INTERFACE OPERATIONAL PROCEDURES.....	25
V. MICROCONTROLLED SEQUENCING.....	33
VI. RING INTERFACE CONNECTIONS.....	35
VII. PROJECT STATUS.....	38
VIII. CONCLUSION AND RECOMMENDATIONS.....	41
APPENDIX 1 - Procedure Flowcharts.....	42
APPENDIX 2 - SMAL Documentation.....	55
APPENDIX 3 - Microcontroller Documentation.....	68
APPENDIX 4 - CRC Specifications.....	78
APPENDIX 5 - Repeater Design proposals.....	80
APPENDIX 6 - RI Schematics.....	88
RING INTERFACE PROGRAM.....	98
BIBLIOGRAPHY.....	109
INITIAL DISTRIBUTION LIST.....	110

LIST OF FIGURES

1. A Sample Data Communication Network.....	9
2. Transmission Encoding and Clocking.....	13 ✓
3. Reception Decoding and Clocking.....	13 ✓
4. Message Format and Transmission Code.....	21
5. Ring Interface Timer Hierarchy Distribution.....	23
6. Receive Handshaking with the Host Processor.....	28
7. Alter PNAME Memory Handshaking.....	30
8. Xmit Handshaking Sequence.....	30
9. Ring Interface Connections.....	37
10. RI to Repeater Control Lines.....	40

ACKNOWLEDGMENTS

The ring network project and the development of a reliable ring interface was undertaken under the direction of Assistant Professor Raymond H. Brubaker, Jr. The author would therefore like to express his gratitude to Professor Brubaker for his insights and patience during the developmental process, without whose assistance this project would have never reached completion. Also, special thanks go out to my friend and fellow IGEP, Andy Pease, for his painstaking efforts in helping to transform schematics into hardware.

I. INTRODUCTION

The data ring communication project at the Naval Postgraduate School has been underway for a year in an effort to establish such a data network at the school. Through the work of Lieutenant Keith Albert Hirt, the project was formulated as described in his master's thesis, "A Prototype Ring-Structured Computer Network Using Micro-Computers." In this report, the conceptual design and introduction to a Distributed Computing System has been presented and will not be included here. For a detailed introduction as to what this system entails, reference should be made to that publication. Only a general summary of these results will be included herein for completeness.

A Data Communication Ring Network consists of a unidirectional serial communication link, interfaces which can connect themselves to this link, and host processors from which and to which data (in the form of messages) can be passed. Figure 1 refers to such a possible network. The two computers, micro-computer, terminal controller, and disc system would be examples of host processors. Each is attached to the ring through a ring interface (RI) which enables them to communicate with any of the other hosts.

The key to the reliability of this type of system lies in the reliability of each of the individual ring interfaces and the method by which they control the operation of the ring. In order to maintain high reliability, no single ring interface (or node) is given ultimate control of the ring. If this were done and the master node were to fail for any reason, the whole system would be totally inoperable until the central controlling node was replaced or repaired. Therefore, to avoid this problem,

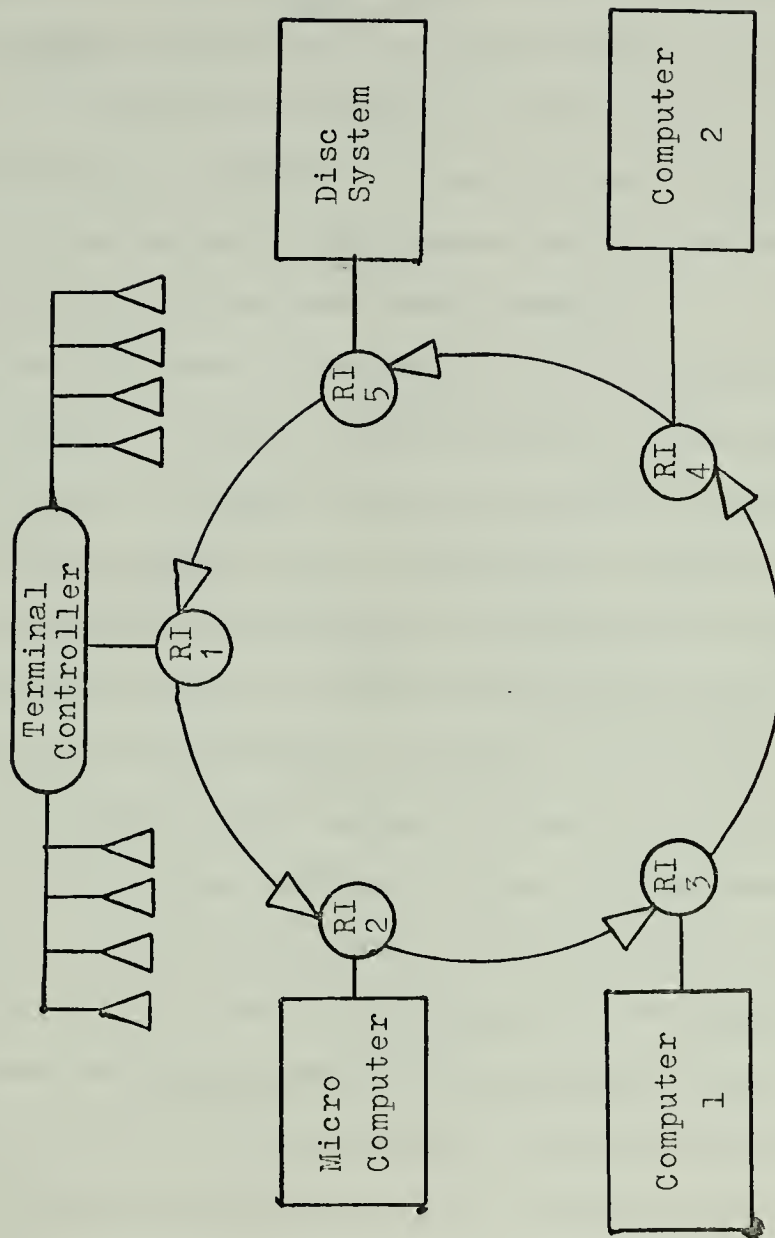


Figure 1. A Sample Data Communication Network

all nodes are given the capability to take charge of the ring. Also, an explicit order, control hierarchy, or "chain of command" is built into the hardware of each node (via a timer) so that a node will take control of the ring only if there is no other node "higher in the chain of command" to do so. This will be explained in detail later in this report. The important thing to understand is that all ring interfaces have the capability to take control of the ring and that only one ring interface will have control at a time under normal operation.

When a node has control of the ring, his host processor is then permitted to transmit a message to another processor in the system. If his host does not wish to transmit, however, the RI then passes control to the next node "downstream" in the ring and begins waiting for messages destined for his host or for control to be handed to him again. When a message arrives for his host, the RI simply signals his host to prepare to receive data, and then begins to deliver it, one byte at a time. At the end of the message, the RI informs his host that he is finished receiving and then continues watching for either another message addressed to his host or an opportunity to take control of the ring. Notice, however, that while a RI is receiving a message from the ring and delivering it to host, he does not remove the message from the ring. Instead, he merely copies it, one byte at a time. This means that the message continues around the ring and may be sent to more than one processor in a single transmit sequence. However, when the message finally returns to the originating node, it is then taken from the ring and the originating node passes control to the next node in the ring.

This then is the basic operation of a ring structured network. Obviously there are many synchronizing problems which have not yet been discussed; however, these are left for discussion later in this paper.

The author wishes only to present the basic operation of a ring network here and to introduce some of the terms which will be used extensively in later sections. If additional information and background is needed, the reader should refer to Hirt's thesis as previously stated.

II. DATA HANDLING TECHNIQUES

As stated in the introduction, the Ring Interface handles all of the receiving and transmitting procedures for his host processor. The host merely delivers to his respective RI one byte of data at a time during transmission and then receives one byte at a time during reception. However, data is shipped serially on the ring and in an encoded form. Therefore, the data bytes must be encoded and put in a serial form by the RI during transmission and then decoded and collected into byte form during reception. These processes will be discussed here.

A. TRANSMISSION

After the ring interface has taken control of the ring and has determined from his host that a message is to be transmitted, he then takes the first byte of data from the host (as shown in Figure 2) and places it into a parallel-in, serial-out shift register. A special transmission clock is then used to shift the data through an encoder and out to the ring. Each of the original data bits is thereby transformed into two transmission bits. (The code for this transmission process is shown in Figure 4.) Notice that with this code, it is impossible to get three zeros or three ones next to each other in a transmission sequence. Also, this implies that the ring must transmit twice as many bits in encoded form than were in the original message. However, since it is impossible for three identical digits to appear next to one another in the encoded serial message, this implies that the receiving RI can watch for this case and thereby detect transmission errors. This is one of the performance tradeoffs made during this project. However, since the network is able

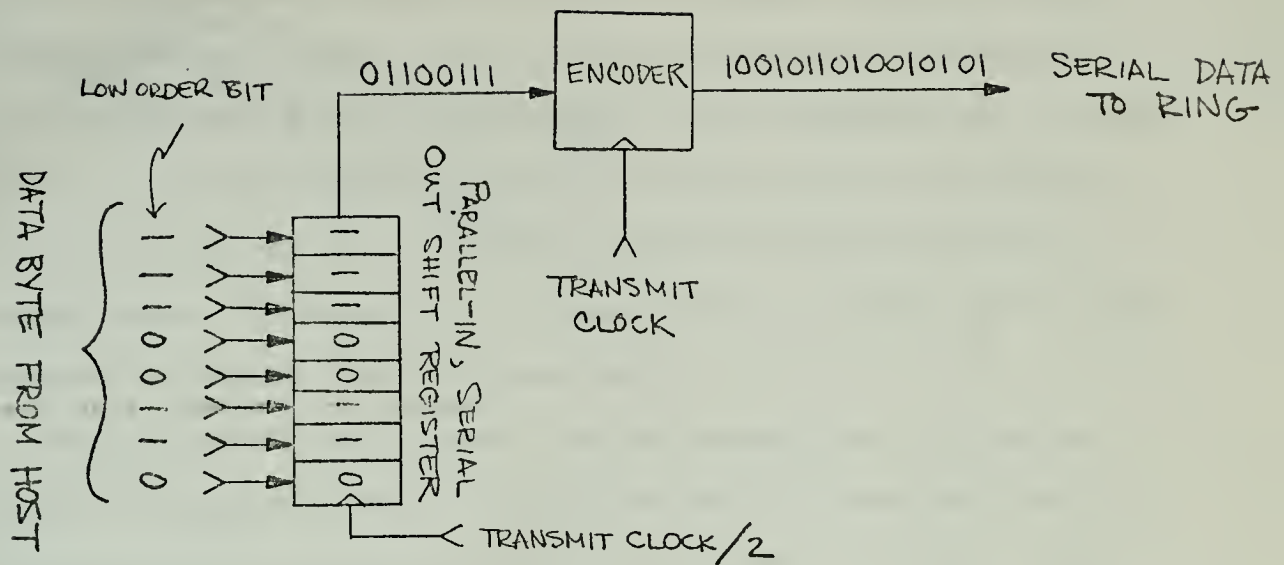


Figure 2. Transmission Encoding and Clocking

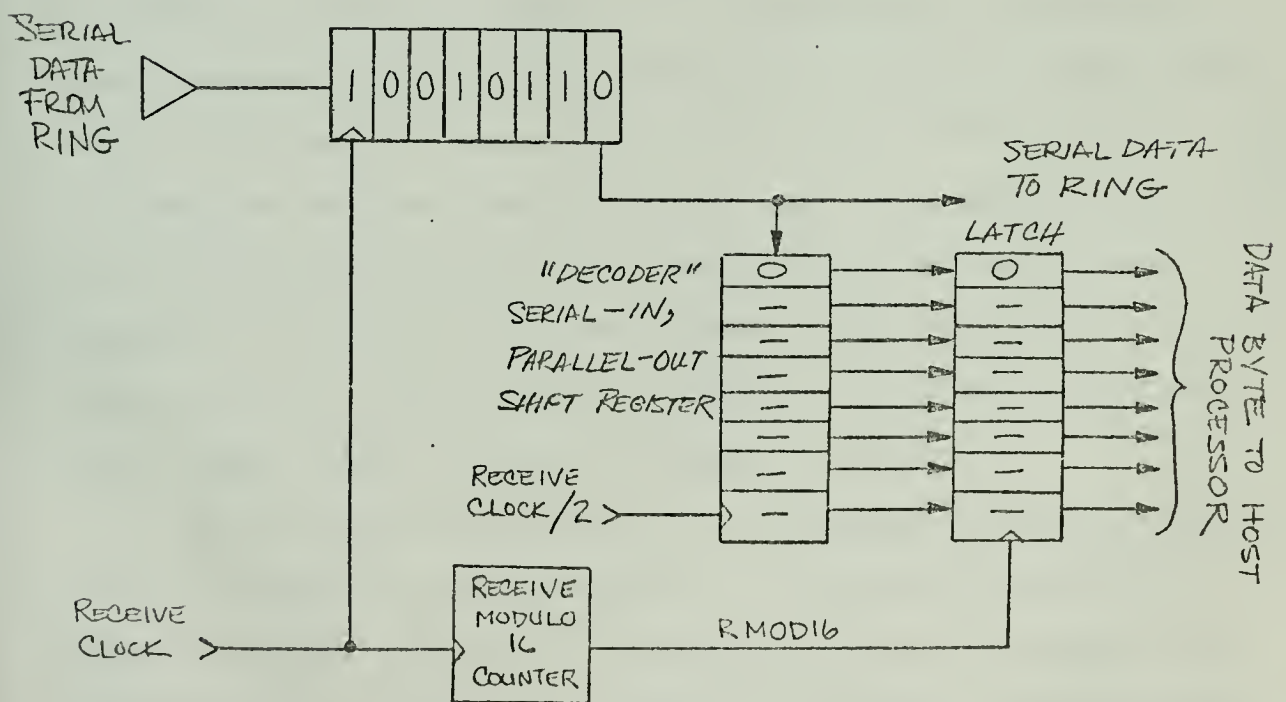


Figure 3. Reception Decoding and Clocking

Is this enough to handle
various holes for transfer
elaborated - see caution
please



to run at a relatively high bit transfer rate (100K to 1000K bits/sec), a substantial data rate is still possible. Furthermore, employing this code also enables the RI to recover the clocking information that is needed to process the data directly from the message itself since only two identical digits can appear together in normal transmission before a change occurs. Consequently, this type of code is called a self-clocking code and was another reason for selecting it.

Finally, notice that the data bytes are encoded from the low order bits to the high order bits. This implies that low order bits leave the RI first during transmission and arrive first during reception. Also, notice that the shift register must be clocked only half as fast as the encoder since the encoder produces twice as many bits as it receives.

B. RECEPTION

During the receiving sequence, the RI transforms the encoded transmission sequence back into a byte configuration for his host while watching for three identical transmission bits in a row. If this occurs, the RI signals his host that there has been an error in transmitting the message (called a Bipolar Violation). During this process, the RI uses a reception clock (recovered from the incoming data) to time the bits as they arrive. When sixteen have passed through a serial-in, parallel-out shift register, they are decoded and passed to the host.

As the bits were being encoded, in the transmission process, the first encoded bit from the encoder was actually the original data bit and the second was merely a "filler" bit designed to aid in error detection, synchronization, and clocking. For example, a zero into an encoder produces an encoded transmission pair consisting of a zero followed by a one. (In other words, the bits come out of the encoder in that order--a zero followed

by a one). Thus the encoded zero is the same as the original zero and the one is merely "garbage" to aid in error detection. Therefore, all that is needed to decode the message is to just "grab" the first bit of a transmission pair and discard the second. Thus instead of actually using a decoder to transform the transmission pairs back into their original form, the RI just takes the leading bit of each transmission pair and places it into a serial-in, parallel-out shift register as shown in Figure 3. A counter is then used to tell the RI when sixteen bits have passed through the preview window (which implies that eight data bits have been assembled in the "decoder" shift register). A latch is then triggered and the newly assembled data byte is taken from the shift register and sent to the host.

Notice that the receive clock cycles for each and every transmission bit. Therefore, this clock is used (as shown in Figure 3) to send data through the preview shift register, to clock the receive Modulo 16 counter, and (by using every other pulse) to drive the "decoder" shift register.

Thus, in summary, the RI transforms the data byte from the host into an encoded serial output to the ring during a transmission sequence and assembles the "real" bits from the transmission data and passes them to the host in byte form during the reception process. Obviously, timing is very important during these processes to avoid missing data or assembling it improperly. For instance, if the "decoder" shift register should slip out of phase by one bit, it would assemble the "garbage" bits and discard the "good" data. Therefore, timing will now be covered in an effort to explain how these possible errors are avoided within the ring interface.

III. CONTROL PROTOCOL

Distributing the control of the ring to each of the ring interfaces has the advantage of increasing the overall reliability of the system. However, it also creates a more complex synchronization and timing problem. Since only one RI can have control of the ring at any point in time to insure proper operation of the entire network, a system of synchronization symbols (tokens) and a timing hierarchy (or "chain of command") relationship was developed between each of the nodes in the ring network. The tokens and "chain of command" constitute the Control Protocol of the system and will now be presented.

A. TOKENS

In order to punctuate the continuous flow of data on the ring, three control tokens were defined. These tokens are shown in Table 1 along with their hexadecimal and binary configurations. Each of these tokens has the high order half-byte in common--1110. This sequence is obviously a Bipolar Violation and therefore could never appear normally in any encoded message. Thus, it is used to trigger circuitry within the RI which enables the low order half-byte to be decoded. If any of the three low order configurations appear (1100, 0101, or 1010), the byte is then recognized as a control token by the RI and the appropriate circuitry is enabled. Care was taken to maximize the Hamming Distance between the low order half-bytes in order to minimize the possibility of ring noise transforming one legal token into another. (The Hamming Distance is computed by performing an exclusive or operation on the control tokens two at a time and counting the number of ones in the result. The higher the Hamming

TABLE 1

<u>Token</u>	<u>Hex Representation</u>	<u>Binary Representation</u>
SOM	EC	11101100
EOM	E5	11100101
CTL	EA	11101010

Note: Control tokens are also shifted to the ring low order bits first

TABLE 2

Flag name	Value	
	0	1
Message Status Bit 1	No RI matched and accepted message	At least one RI matched and accepted the message
Message Status Bit 2	No RI matched without accept- ing	At least one RI matched but did not accept
Message Status Bit 3	No target RI recognized a CRC or BPV error	At least one RI recognized a CRC or BPV error

Distance, the lower the probability of transforming one token into another). Out of a maximum of four, each of the control tokens is at least two Hamming units from the others.

Thus after defining this system of punctuation, the format for the message transmission was defined. Figure 4 represents a typical message. The different segments and their meanings will now be discussed.

1. SOM--This is the Start-of-Message token which is used to tell all receiving RI's that a message is to follow. Since all of the counters within the ring interfaces work on a modulo sixteen basis, eight zeros are added to the end of this token in order to pad the length to sixteen.

2. PNAME--The Pname (Process Name) segment is used by the Ring Interface to determine if the message is intended for its host. Each host offers various processes to the system. The presence of a process within the host is recorded in a 256 by 1 RAM (Random Access Memory) which resides within the RI. When a message is sent to the ring and destined for a certain process, each RI checks its memory to see if his host offers that service. If a match occurs (a "one" located in the memory location which corresponds to the decoded PNAME byte), the message is then relayed to the host for processing. If it does not match (a "zero" in that location), the message is ignored.

3. Message Body--The message body consists of the sixteen bit encoded segments (eight data bits) which are to be relayed to another processor in the ring network. The messages, therefore, are variable length. However, to avoid the situation where one processor takes control of the ring through his associated RI and keeps it for the rest of the day while transmitting vast data banks through the system (for instance), a maximum length is defined (via the timer), expiration of which causes the

What happened because of
it, is it not?
Will that way be sent?
No
unless it is for nothing?

"hogging" host's RI to shut down and exit from the ring. This maximum length is implementation dependent and may be varied from implementation to implementation.

4. CRC Segments--These two sixteen bit segments (two eight bit data bytes) are generated by each RI as an additional transmission error checking technique. Basically, the body of the message is considered to be one huge number and a polynomial technique is used to divide it. The remainder is what makes up the CRC segments and is placed after the main body of the message. Then when the message is received, the message is again divided, but with the CRC bits added in. This time the remainder should be zero when subjected to the polynomial technique. If it is, the message has been transmitted correctly--if not, the receiving host and originating host are informed that a CRC error has occurred.

5. EOM--The End-of-Message token with its associated eight bits of padding follows next and serves three useful functions. First, it tells the RI that there is no more information to relay to his host. Secondly, it signals the CRC circuitry to check its remainder for a transmission error. Finally, it is used to delimit the message body from the node address and status information.

6. Node Address--The sixteen bit node address is placed onto the message so that the originating RI can insure that his message has returned to him properly and that no ring error condition exists. Each node has a unique node number which therefore limits the maximum number of nodes on the ring to 256 since the eight bits must be encoded for transmission to prevent a bipolar violation from being detected.

7. Status Bits--The three status bits which follow the node number are used by the host to determine if his message was received correctly

by the target RI's. The information that is generated from these status bits are summarized in Table 2.

Therefore, the SOM and EOM are used to punctuate the message so that the RI can separate host data from the data it must have to keep the ring operational (like node number and status information). The final token, the CTL or Control token is used to pass control of the ring from one node to another. When a RI receives a CTL token, it is allowed to take control of the ring and place a message on it. When it has completed transmission (or if the host does not wish to transmit), it places another CTL on the ring which is then received by the next node in the ring.

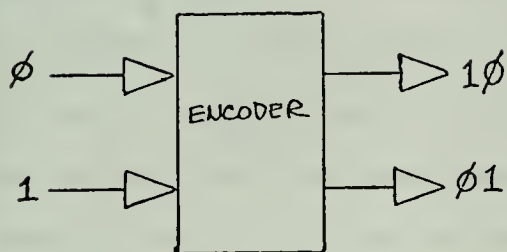
In summary, these three tokens serve as the punctuation needed to keep the ring operational. Taking these from the system would result in the same chaos as would occur by taking all the punctuation, spaces, and uppercase letters from this report. it would be entirely impossible to tell where one message stops and another starts

B. TIMING HIERARCHY

Through the use of the control tokens the ring network can be maintained in a steady state condition. But error condition handling and initial power up has not been discussed. The question of how the first CTL arrives on the ring and who takes command when a ring error condition exists will now be addressed.

As previously stated, a ring interface can take control of the ring network whenever a CTL token arrives at that node. However, it can also take control if it feels that something is wrong with the ring and that it has waited "too long" without seeing either a SOM or CTL. This is the "time-out function" of the RI. In order to insure that only one RI "times-out" and takes control of the ring at any point in time, a timing hierarchy

Encoding Scheme



Note: When a zero enters the decoder, a zero exits followed by a one. Thus the first bit of an encoded transmission pair is the same as the data bit which produced it.

MESSAGE FORMAT

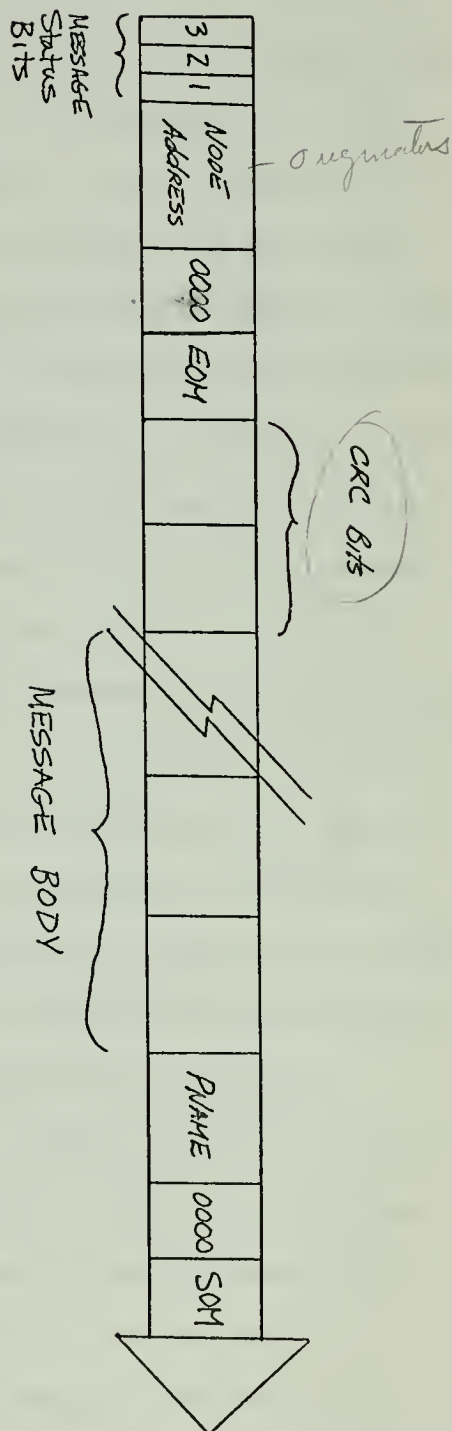


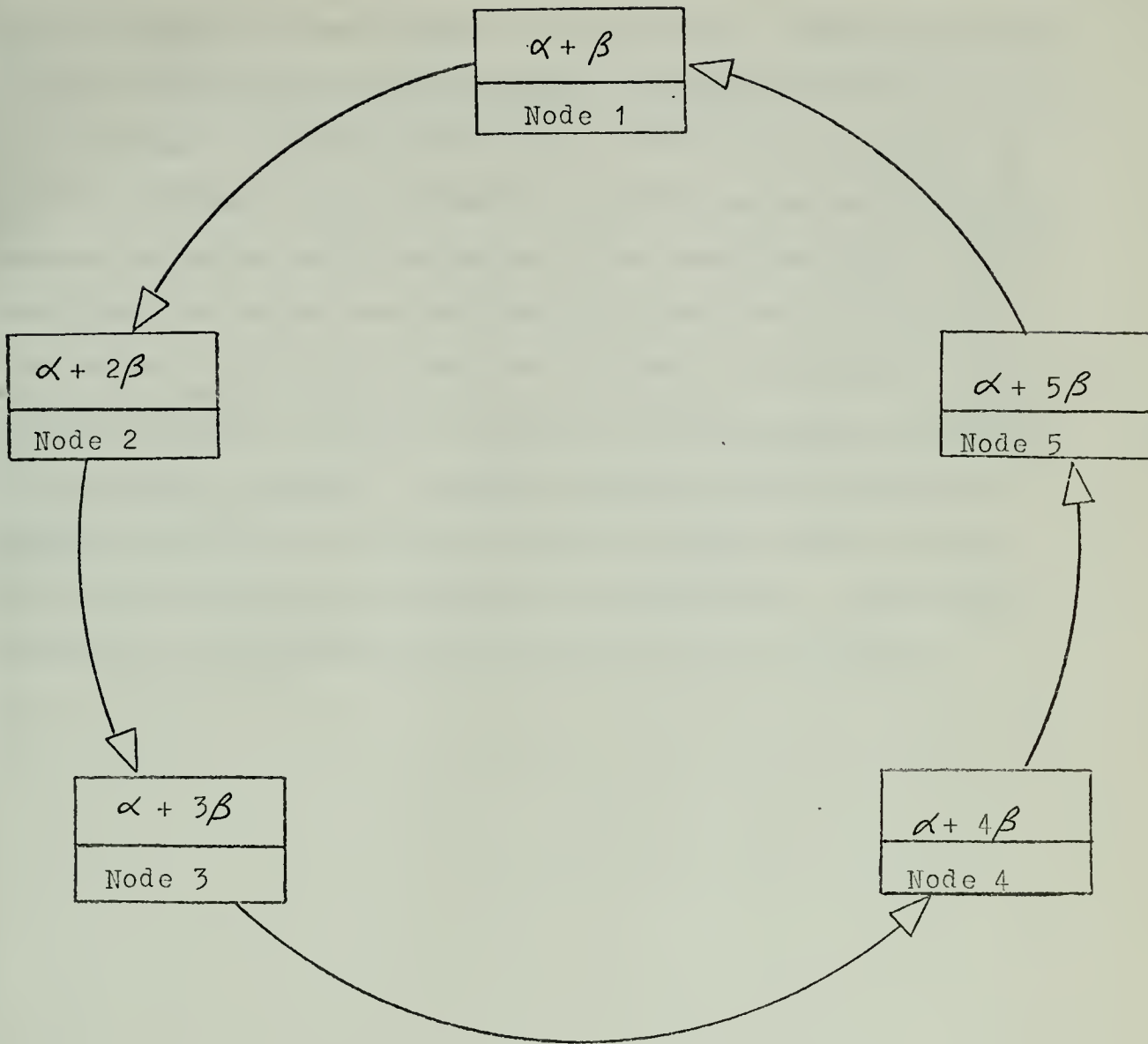
Figure 4. Message format and transmission code

2.

Note - I probably have
particulars of the
a long / the end.

was implemented and a unique timer built into each RI. (Refer to Figure 5) α is the maximum time allowed for a node to transmit. After that amount of time, the originating RI shuts off his "hogging" host and exits from the ring. β is the incremental period of time that will insure that only one node can "time-out" at any point in time and must be strictly greater than the maximum amount of time needed to send a bit around the ring-- γ . If an error condition occurs and the CTL token is somehow lost from the ring--all of the nodes begin a waiting race to see who will take control of the ring and place a new CTL on it. The RI who has the "shortest timer" (smallest $\alpha + n\beta$) and is still connected to the ring will "time-out" first. This is a relative question. If, for example, nodes 1 and 2 in Figure 5 were not hooked up to the ring, node 3 would have relatively the shortest timer and would take control and send a CTL to the ring. Thus, no one node is given the responsibility to either start the ring or correct an error situation. Also, during initial power-up, one or more nodes may enter the ring and begin waiting for an opportunity to take control and transmit. However, since no node has control of the ring, the waiting race will begin again and the node who is highest in the "chain of command" (i.e. has the shortest timer) will take control and transmit a CTL token and the ring will again resume steady state operation.

This timing hierarchy then, allows nodes to enter and exit from the ring at will without interfering with the overall operation of the ring. (The ENTER and EXIT procedures will be discussed in detail later.) If, at any point in time, a CTL does not reside on the ring and no RI is in control, the timing hierarchy will provide a "command" node to restart the system. Also, if two CTL tokens are detected on the ring, a ring error condition is recognized and the detecting RI simply removes the extra token from the ring and begins waiting. This insures that two nodes will



- 1) α = maximum time allowed for message transmission
- 2) β = the incremental time delay
- 3) γ = the time required for a bit to travel around the ring when all nodes are connected
- 4) Requirement: $\beta > \gamma$

Figure 5. Ring Interface Timer Hierarchy Distribution

not be attempting to send messages at the same time. (Further explanation of ring error procedures will be discussed in detail later in this report.)

In summary, the three control tokens and the "time-out" hierarchy enable the individual ring interfaces to recognize messages and pass control between one another while insuring that no two nodes take control at the same time. The unique timers which are built into each RI also enable the network to detect error conditions and restart automatically. Also, the timers regulate the amount of time that a host processor can transmit to prevent ring "hogging." The actual procedures which are utilized to transmit, receive, enter and exit the ring, and detect error conditions will now be presented and discussed in flow charted form. Actual implementation of these procedures will be discussed in later sections.

IV. RING INTERFACE OPERATIONAL PROCEDURES

In an effort to simplify the complexity of the data handling sequences required to maintain normal operation of the ring while detecting and correction error conditions, the required microinstructions have been functionally grouped into procedures. These include INIT, MAIN, XMIT, RECEIVE, XRINGERR, RRINGERR, ENTER, EXIT, EOMWATCH, XMITERR, and DIE. Flow charts are included in the appendix and a description of each procedure and its contribution to the overall system is presented here.

INIT--The initialization procedure is used to place the RI in a ready state before it actually attempts to enter the ring. In this procedure a flag is set to tell the RI that it is not yet connected to the ring. Also, the "relay" mode is selected so that all information that enters the ring interface will be echoed to the ring. This insures that no information is lost from the ring while nodes enter and exit the system. INIT is activated by resetting the RI.

MAIN--This procedure is the heart of the system. From here the RI watches for the CTL and SOM tokens which indicate that control will be passed to the XMIT or RECEIVE procedures respectively. However, MAIN also performs three additional functions. It checks to see if the RI is connected to the network. If not, the system is placed into a wait loop until the host processor signals the RI to connect itself to the system. Secondly, while watching for the CTL and SOM tokens, the RI also monitors the host to see if he wishes to disconnect from the system. If so, the EXIT procedure is called and the RI exits from the ring. Finally, in MAIN, the timer is monitored. If no SOM or CTL token passes the RI's

for the first
year for the
first year

2 1/2
not much
but 100 lbs. off
handing over
issue new etc

preview window within the built in timer limit, the RI assumes control of the ring and places a CTL token on it. It then resets the timer and reenters the waiting loop.

Thus, from this procedure the control tokens are detected and appropriate subroutines are enabled. Also, the ring error conditions are detected and RI control is assumed. Finally, exit from and entrance to the network is enabled and monitored.

ENTER--The ENTER routine is used to electronically enter the RI into the network. In this procedure, all the status flags used within the RI to detect the occurrence of events are reset. The timer is also reset and the RI is placed into a loop waiting for a CTL, SOM, or the timer. Basically, the RI is waiting for the opportunity to enter the system without interrupting any existing messages on the ring. Thus, the RI waits until a CTL token is detected (or the timer expires which indicates that no one is in control of the network) before attempting entrance. When the CTL appears, the RI simply waits until it passes and then connects itself to the ring. The appropriate status flag is then set to let the RI know that it is connected.

EXIT--The EXIT procedure is merely the inverse of the enter routine except for one important point. When the CTL token is recognized, it is taken from the ring (or "gobbled"). This places the ring in the error state where no RI is in control. Thus, the RI with the shortest timer still remaining on the ring will assume control and replace the missing CTL. This mechanism prevents interference with message traffic by an exiting RI.

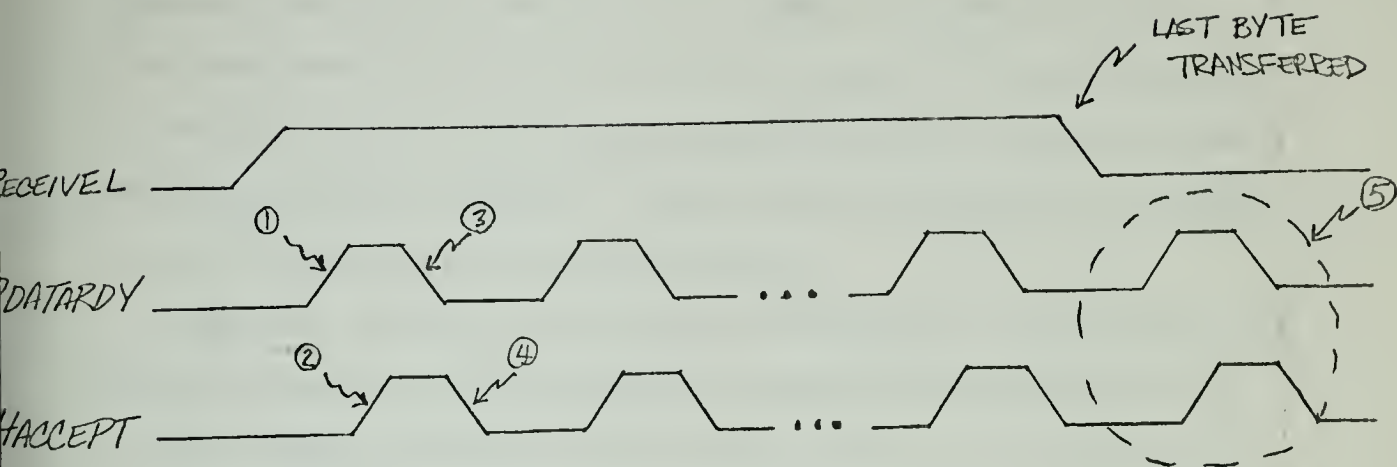
RECEIVE--The RECEIVE routine contains the sequences required to recognize a message in the format shown in Figure 4. As the message enters

the RI, it is checked, (the Pname byte is processed and checked against the RI Pname memory), assembled, and transferred to the host byte by byte. Care is taken to insure that a message overrun does not occur during this process. If the host does not accept the data byte before a new byte shifts into the "decoder" shift register, a data overrun occurs and the byte of data is lost from the message. When this occurs, the host processor and the originating RI must be informed that the message was not received properly. Therefore, when an overrun is detected, the Receive overrun flag is enabled.

The "handshaking" which is required to hand a byte of data to the host is shown in Figure 6. Notice that the host is required to positively acknowledge that it is ready to accept data and again that it has copied the data. This is done to insure that the message is properly received by the host.

After the message is received, the three message status bits are modified on the returning message (according to the protocol established in Table 2) in order to inform the originating RI of the status of the message reception. When this is complete, the MAIN procedure is activated and the RI begins the waiting process again. If, however, during reception an extra CTL or SOM is detected, or if the timer expires, this indicates that a "Ring Error" condition exists and the RRINGERR routine is executed.

RRINGERR--This routine is used to restart the ring from an error configuration. It implies that during reception of a message, the RI has encountered a misplaced CTL or SOM or that the timer has expired indicating that no one is in proper control of the ring network. Therefore, a Ring Error flag is enabled to tell the host what has happened and the RI



When the RI receives a message for its host, it raises the Receive Line to tell the host to prepare to accept the message

- 1) When the data byte has been assembled, the RI raises the Receive Data Ready Line (RDATARDY)
- 2) The host raises the Host Accept (HACCEPT) line to indicate it has accepted the data byte
- 3) The RI drops the RDATARDY line indicating end-of-transfer
- 4) The host drops HACCEPT acknowledging end-of-transfer
- 5) This last "handshake" is used to transfer the message status to the host. At this time the host checks the status flags to see if the message was received properly.

Figure 6. Receive Handshaking with the Host Processor

then removes the next CTL from the ring and returns to the MAIN routine. Removing the CTL is the solution for a number of synchronization problems which could occur if two or more RI's placed CTL tokens on the ring at the same time. If no CTL arrives within the timer limit, the RI merely returns to MAIN immediately. Notice that the Receive line is dropped to tell the host he is no longer receiving.

XMIT--The transmit routine performs two functions. It is used by the host to record active process names in the RI Pname memory and is also used to transmit messages through the ring network. If the host wishes to enter or clear a process name from Pname memory, it raises the Alter line to the RI, as shown in Figure 7. It then places either a one (if it wished to enter a process) or a zero (for clearing a process) on the PNAME ACTIVE line. The RI raises the Demand line when ready to enter the name into memory and the host places the eight bit address of the location to be modified into his output buffer and drops the Alter line. The RI then enters the data on the Pname Active line into the memory location indicated by the byte in the host input buffer and loops around to see if the host wishes to modify another location (the host has approximately 10 microcontroller cycles to raise the Alter line if it wishes to enter or delete more process names). If the Alter line remains low, the RI places a CTL on the ring and returns to MAIN. (Note that the host should clear all memory locations immediately after initial entry to the ring since the initial power up leaves the Pname memory in an undefined state.)

If the host does not wish to modify Pname memory, it can request to transmit by raising the Xmit line. This can be done at any time and the RI will service the request upon the receipt of a CTL token. However, if a message arrives for the host (i.e. matching a name in Pname memory)

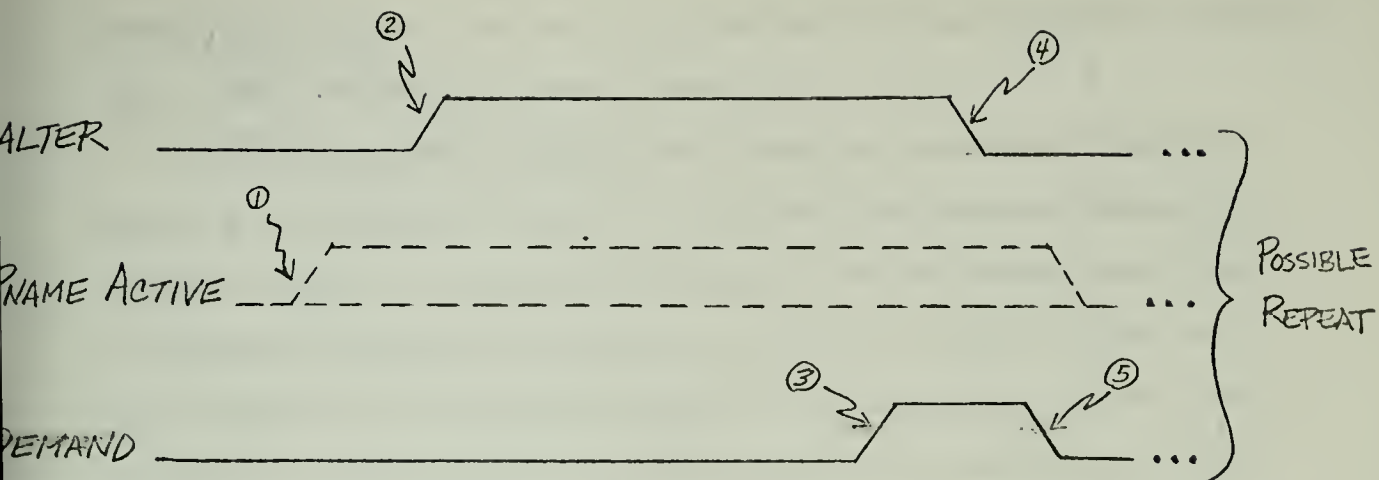
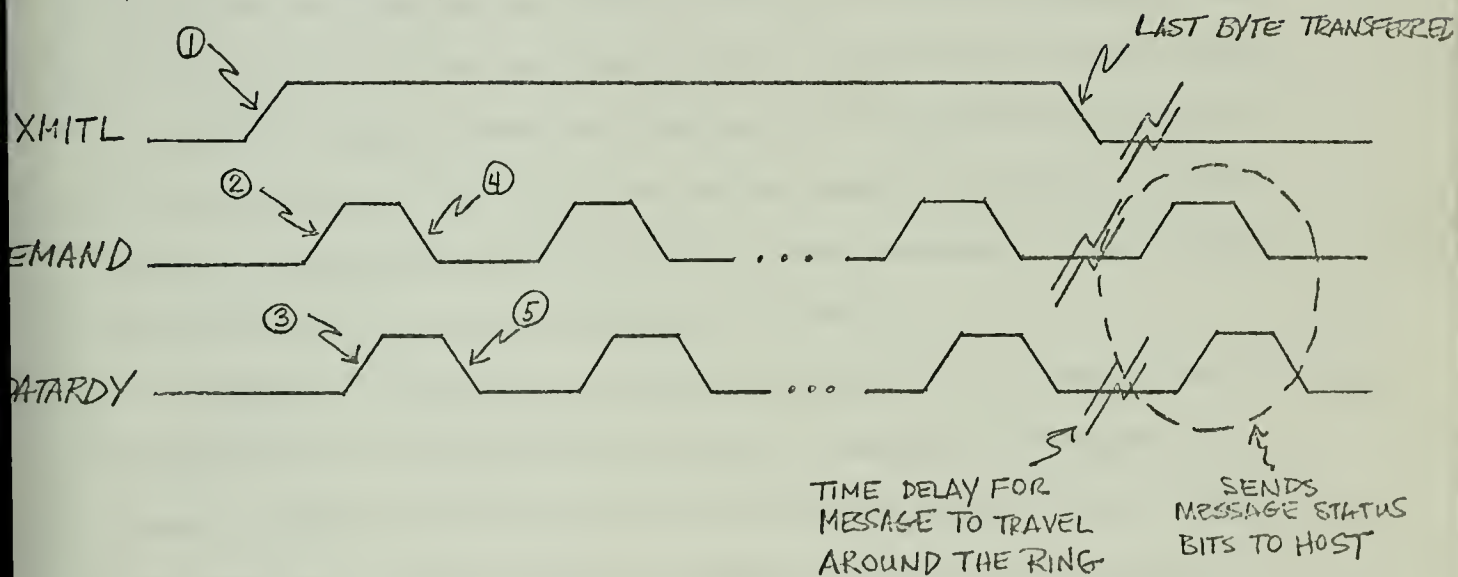


Figure 7. Alter PNAME Memory Handshaking

(Case 1-- normal)



(Case 2-- incoming message preempts transmission)

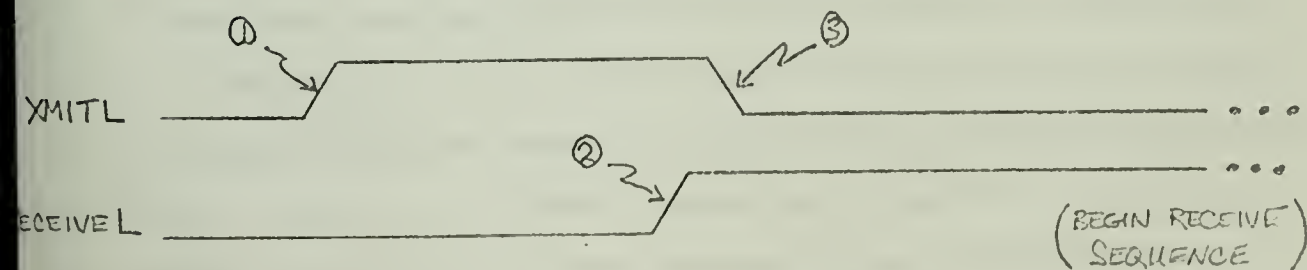


Figure 8. Xmit Handshaking Sequence

before a CTL arrives, the host drops the Xmit line and prepares to receive the incoming message (as shown in Case 2).

In the transmission phase of XMIT procedure, the message format in Figure 4 is constructed. After a 32 bit delay (to separate messages on the ring in case of timing-phase differences between transmitting RI's) a SOM token is shifted to the ring. The host then begins transmitting his message to the ring through the RI by the handshaking procedure shown in Figure 8. Note that the first byte of data (the destination Process name) corresponds to the Pname information previously discussed. After the message has been "handed" to the RI, the host drops the Xmit line (as shown) to tell the RI that the message is complete. The RI then enables the CRC circuitry and shifts out the 32 bit CRC information. The EOM and the Node Number are added followed by three zeros which serve as the initial message status bits via the EOMWATCH procedure. This procedure also adds a CTL to the ring so that another node can assume control of the network and is therefore executed immediately after the CRC bits are sent to the ring via the XMIT procedure.

EOMWATCH--The purpose of the EOMWATCH procedure is to enable the RI to watch for the return of the EOM from the message just transmitted. This is a critical timing area since the ring might be very short (only one node connected in the trivial case) and the message could return almost instantly. Therefore, this procedure checks to see if the EOM has already been detected while the RI was busy transmitting the Node Number and Message Status Bits. If the EOM has already been detected, then the Node Number and status information are immediately available for processing. If EOM has not yet been received, the RI waits for its return, checks to see if the node number matches its own (to insure the proper message is returning), and then passes the Message Status Bits to the host.

In either case, if the Node Number fails to match, a ring error situation is detected and the XRINGERR routine is executed. Also, if the host fails to provide data to the RI during transmission in sufficient time to shift it to the ring, then a transmission overrun occurs and the XMITERR routine is employed. Also, as previously discussed, if the host attempts to transmit messages of length (actually, time) greater than the maximum allowed, the RI uses the DIE routine to exit from the ring and shut down operation. These three routines will now be discussed.

XMITERR--This routine is used by the RI whenever a transmission overrun occurs. The transmitting RI merely places eight ones on the ring (which will cause a Bipolar Violation when received by the target RI) to destroy the message, sets a transmit overrun flag for the host, and adds the normal ending to the message via the EOMWATCH routine.

XRINGERR--This procedure is similar to the RRINGERR procedure previously discussed except that it sets a special flag for the host to indicate the error was detected during transmission. This tells the host that either the Node Number which returned was not his, an extra SOM or CTL was detected while receiving back his transmitted message, or that the timer went off while awaiting return of the transmitted message.

Look at this

DIE--This routine is the trap procedure used to keep hosts from "hogging" the ring. Whenever the host attempts to alter Pname memory or transmit messages for a period of time longer than the maximum allowed, this routine is enabled and the RI exits from the ring (as in the EXIT procedure) and then begins an infinite waiting loop. The only way to exit this procedure is to reset the RI manually which executes the initialization routine.

Thus, using these procedures, the RI controls both normal and abnormal operation of the ring. The discussion will now turn to the method of employing these sequences within the RI hardware.

V. MICRO-CONTROLLED SEQUENCING

As presented thus far, the NPS Data Communication Ring does not differ substantially from the Distributed Computer System investigated by Hirt [1] at the University of California at Irvine. Both systems employ a message format, send these messages around a ring to target nodes, and use similar processes to receive and transmit messages. However, though the systems differ in implementation and design, the one radical difference between the two networks derives from the manner in which the control procedures are implemented. In the Irvine system, all sequences were hard-wired into the RI design using state diagrams and sequencing logic. Therefore, in order to change the method or order in which messages are processed, costly hardware modifications must be employed.

To avoid this inflexibility and simplify design, the NPS ring interface incorporates a general-purpose microcontroller which was developed by Assistant Professor Brubaker with the author to generate the desired sequences. Assistant Professor Gary A. Kildall, developed an assembly language for the controller intitled SMAL (Symbolic Microcontroller Assembly Language) and the control procedures for the RI were written in this language. (Appendix 2) This language is operated on the Intellec-8 microcomputer developmental system. Descriptions of the controller and of SMAL are included as appendices to this thesis and will not be presented here.

Through the use of this microcontroller and the SMAL language, the sequencing procedures have been implemented and recorded within the microcontroller on PROM (Programmable Read-only Memory) chips. The program used to generate these sequences is found at the end of this report along

with the generated machine code. The program is well documented via comment statements in an effort to simplify the correspondence between the flow charts and the SMAL implementation of them. Since the microcontroller employs a polling scheme to detect the occurrences of events within the RI circuitry, it is sometimes necessary to execute several instructions between data bit clocking. This requirement implies that the microcontroller must run faster than the data rate. The critical area in the program is found in the RECEIVE procedure. After the Node Number passes, the RI must shift the Message Status Bits immediately to the ring. However, before this can be done, three instructions must be executed. Therefore, the microcontroller is required to run at least four times faster than the data rate to insure proper operation of this procedure.

The remainder of this report will consequently be used to define, explain, and enumerate the control lines required between the RI and the host processor.

VI. RING INTERFACE CONNECTIONS

In order to formalize the interfacing connections required for a host to connect to the NPS Ring Interface, the following section is included. Most of these lines have already been discussed, but are summarized here for completeness. Figure 9 represents this summary.

Receive Group--The RECEIVEL, RDATA RDY, and HACCEP lines are used during message reception to deliver the bytes of information from the RI to the host. The actual data is transmitted over the eight bit data bus from RI to host. The actual handshaking procedure employed is shown in Figure 6 and will not be reiterated here.

Xmit Group--The XMITL, DEMAND, and HDATA RDY lines are used during transmission of a message to the ring network. The transmit sequence is defined in Figure 8. Data is passed to the RI from the host over the appropriate data bus.

Local Command Group--The ALTER and PNAME ACTIVE lines are used during Pname memory modifications as shown in Figure 7. RESET is used to start the RI operation during initial power up and to cause an exit from the Die routine. DCT is used by the host to cause the RI to exit from the ring network.

Status Flags

1. RCRC--When enabled, this flag tells the host that a CRC error was detected during message reception.
2. ROVER--This flag indicates that a data overrun occurred during message reception.
3. XCRC--This flag implies that Message Status Bit 3 returned to the originating RI in the "one" state which indicates that a CRC error was detected by the target RI during reception of a message.

4. XOVER--When enabled, this flag indicates that a transmission overrun occurred during transmission of a message. This indicates that the host did not provide data to the RI fast enough to be shifted to the ring normally.

5. MSB1 and MSB2--These are the message status bits which returned after message transmission. Interpretation of these flags is shown in Table 2.

6. RRERROR--When enabled, this flag indicates that a ring error condition was detected during reception.

7. XRERROR--Similar to RRERROR, this flag indicates that a ring error condition was detected during transmission.

8. DCTD--When enabled, this flag implies that the RI is presently disconnected from the ring.

Note that the Xmit flags remain valid after transmission of one message until transmission of the next.

This, then is the interpretation for the Ring Interface Connections and status flags needed during RI operation. The arrows in Figure 9 indicate whether they are inputs to the RI or outputs from it. The names used in the above description are identical to those employed in the RI SMAL program.

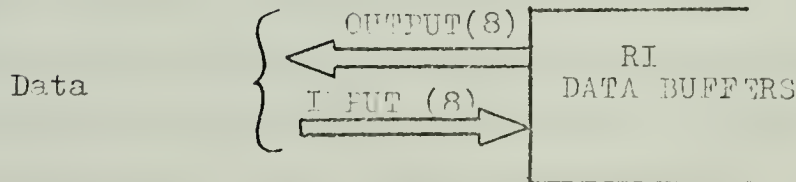
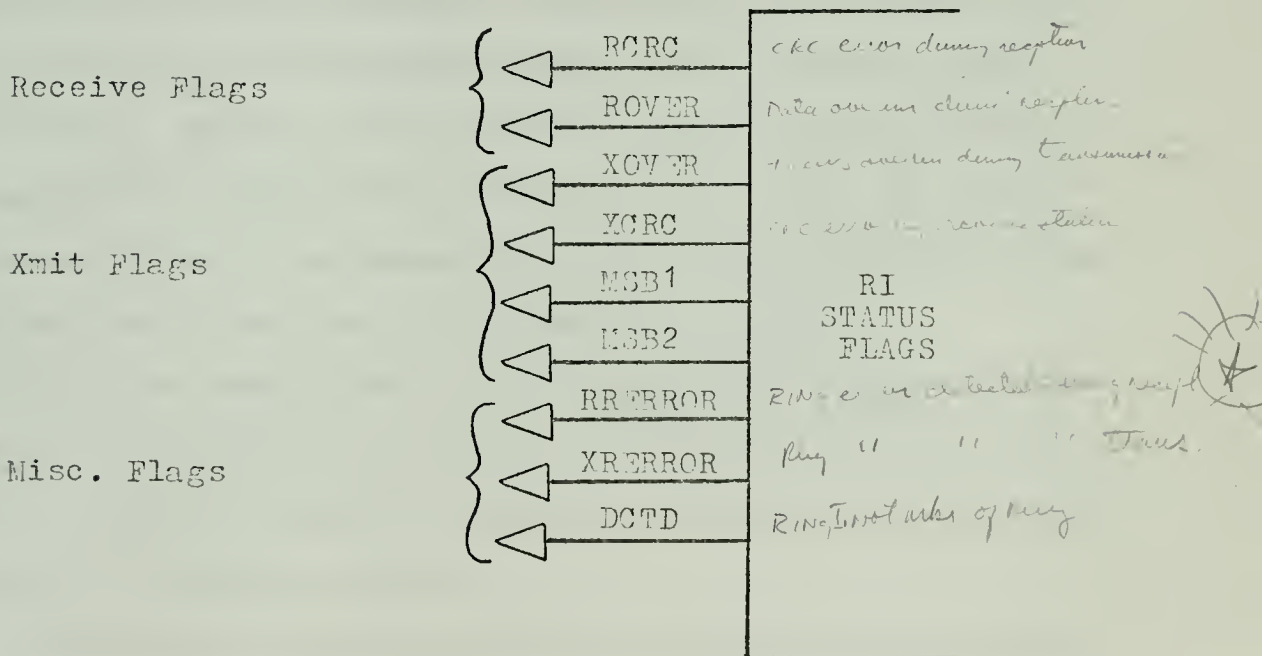
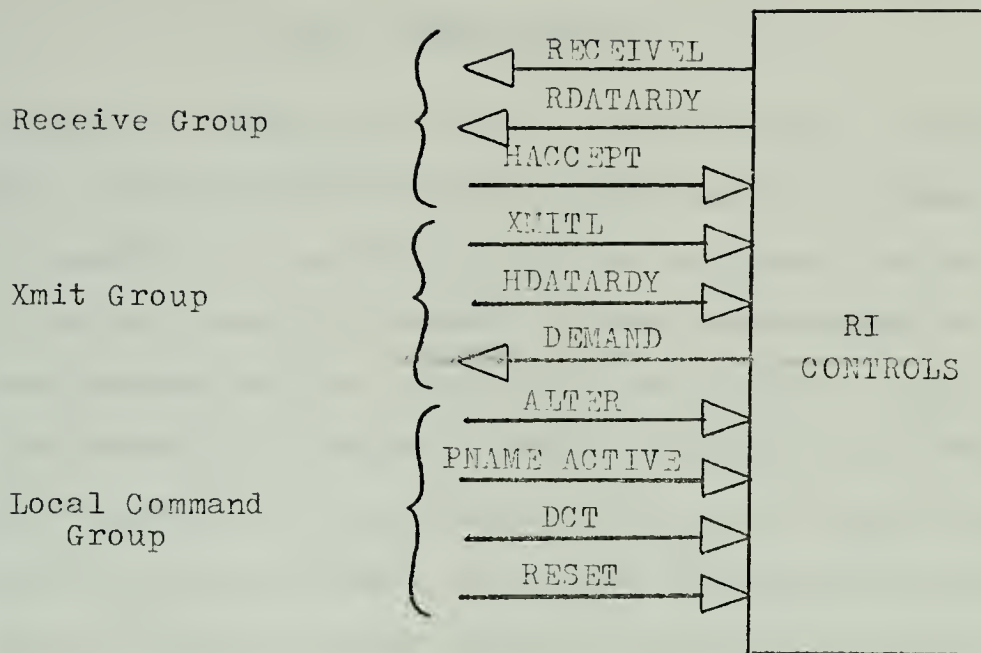


Figure 9. Ring Interface Connections

VII. PROJECT STATUS

The NPS ring network is not a fully developed system. The interface has been constructed and hand tested to insure that the procedures operate as described. During this low speed testing process, the input control lines were connected to manually operated toggle switches so that the host processor could be simulated by a human operator. Also, since the timer mechanism was designed to operate at high speeds with short "time-out" intervals, it was necessary to control this function manually.

Low speed operation offered the advantage of facilitating circuitry and software debugging; however, the full capabilities of the system consequently have not been established. Furthermore, the CRC checking function of the RI is to be implemented using a single integrated circuit from Motorola. However, the LSI chip has not been made available from the manufacturer. Consequently, the CRC circuitry has not been tested. (Design specifications and documentation for the proposed CRC IC has been distributed and is included in the appendix for reference.)

Finally, the repeater which is used to amplify and circulate the messages from the RI has not been designed. Control lines have been established, (as shown in Figure 10), and reference material on proposed implementation is available in Appendix 5.

In summary, the RI has been tested only at low speeds due to the unavailability of hosts capable to interface directly to the system. Future tests should include timer and CRC operation along with high speed data processing before capabilities and limitations can be fully understood. Finally, repeater implementation must be completed and tested to provide

the necessary power to send the data over the distances required and the recovery of clocking information from the incoming signal.

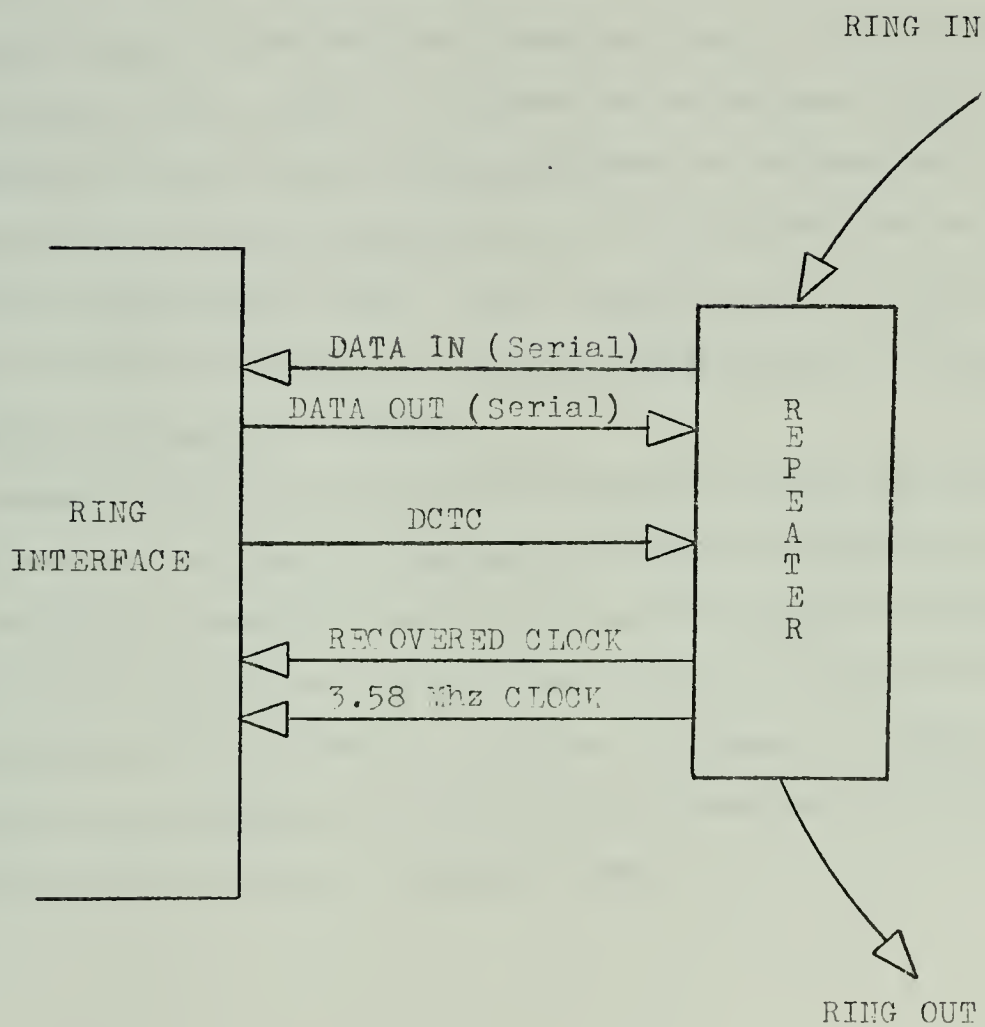


Figure 10. RI to Repeater Control Lines

VIII. CONCLUSIONS AND RECOMMENDATIONS

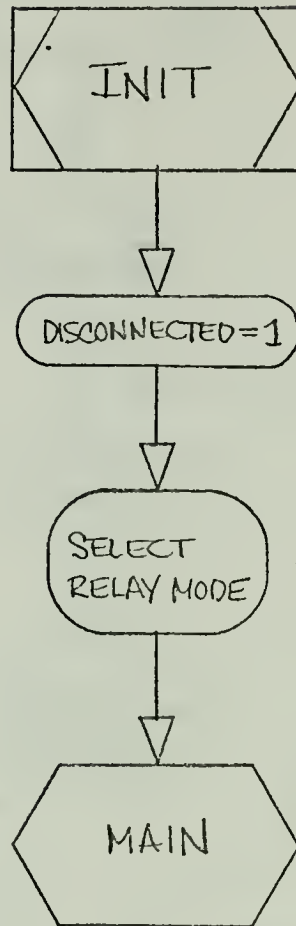
In summary, a prototype ring interface has been designed and constructed which offers flexibility, low cost, and the reliability needed to operate efficiently within the constraints proposed by Hirt. Through the use of a general purpose microcontroller, future modification to operating procedures are not only feasible, but economical and software oriented. The structure of the unit enables modular expansion of the system up to 256 nodes with a linear cost expansion curve. (The estimated cost for each RI will be approximately \$1000.) Also, through the use of Fusable-link ROM vice the PROM technology now employed, higher speeds in the range of 1 million bits/sec seem feasible.

Recommendations are centered around testing. Although the unit has been tested at low speeds, a high speed, full scale testing must follow before the full capabilities and limitations of the system can be known. This leaves the field open for the research which may reveal more efficient data handling procedures. As of now, the system is a working prototype and must therefore be subjected to the normal testing and modifications inherent in such an experimental unit.

APPENDIX 1

RI PROCEDURAL FLOWCHARTS

INIT PROCEDURE

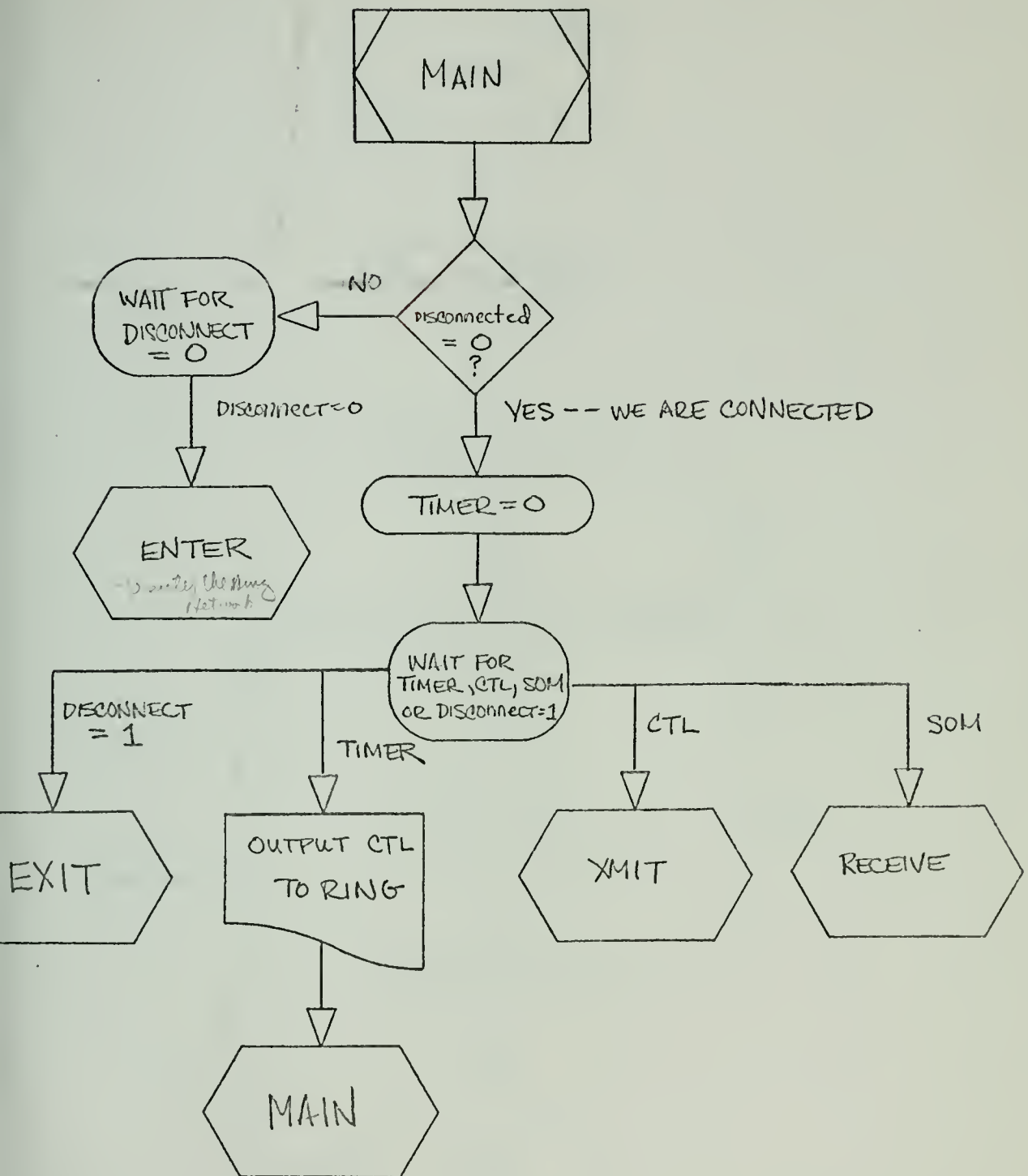


- means on new disconnected from 1-2

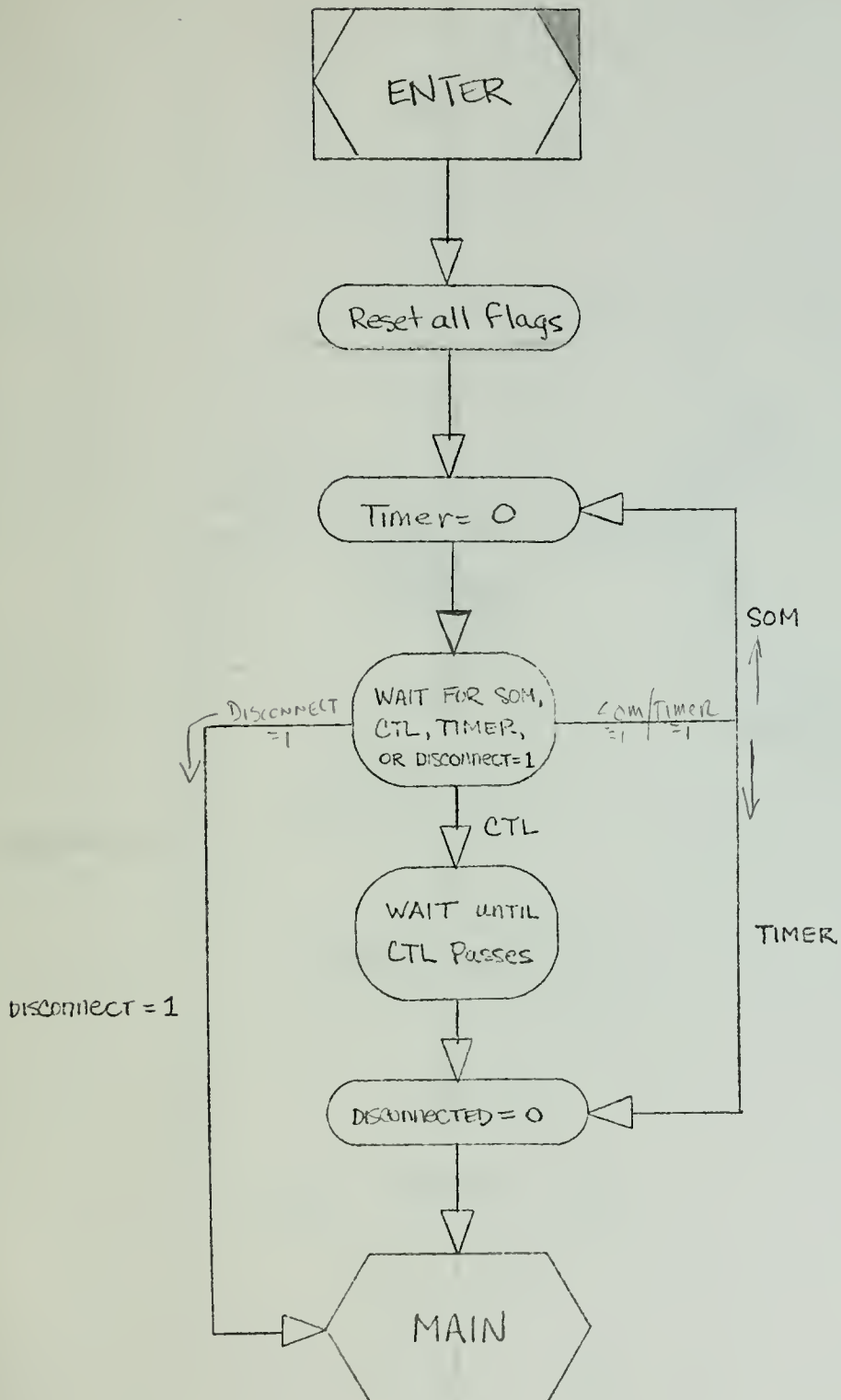
- 1-2 is not a true all data line
- 1-2 is not a true all data line

* NOTE: Resetting the MicroController CAUSES THIS ROUTINE TO BE EXECUTED

MAIN PROCEDURE

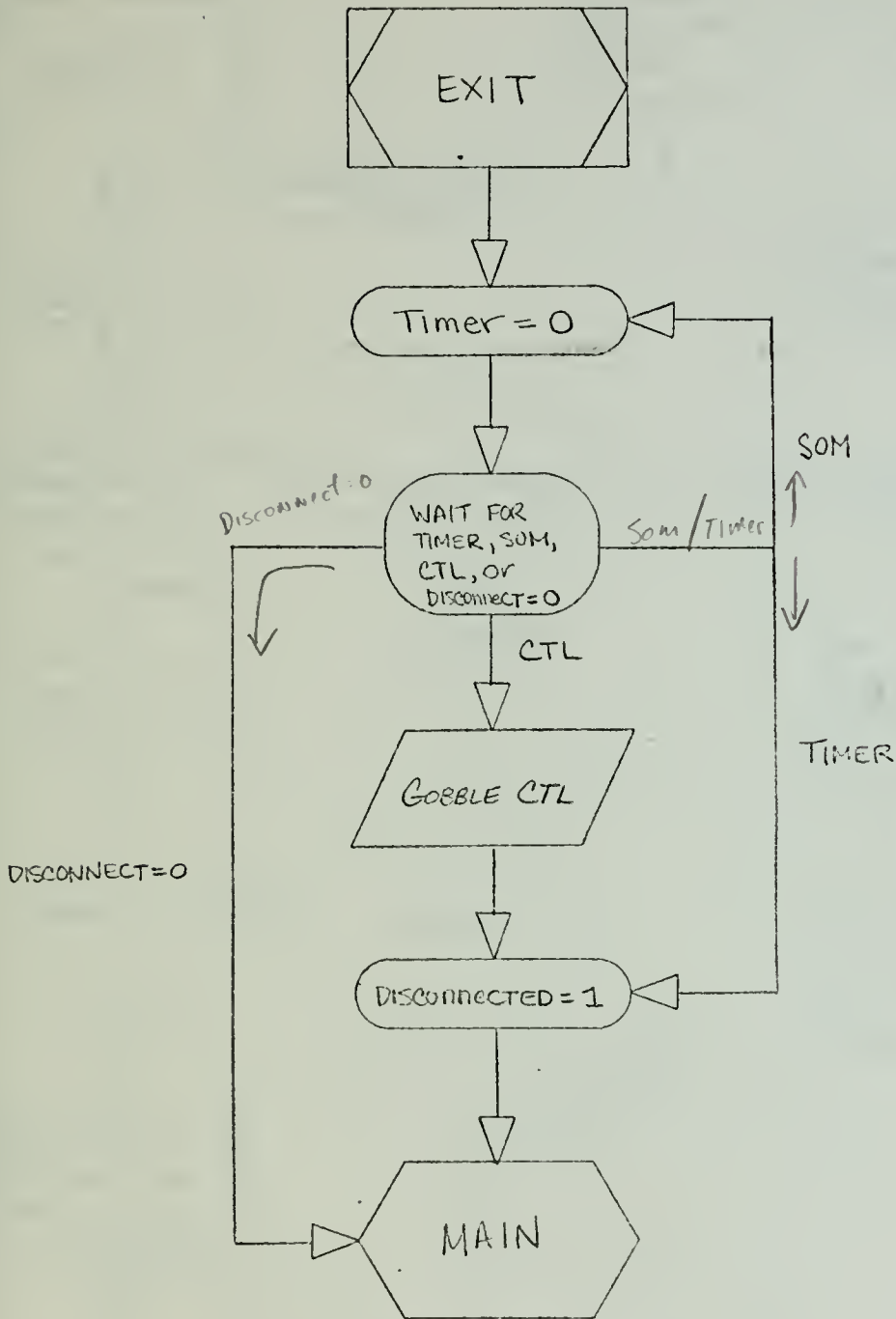


ENTER PROCEDURE



EXIT PROCEDURE

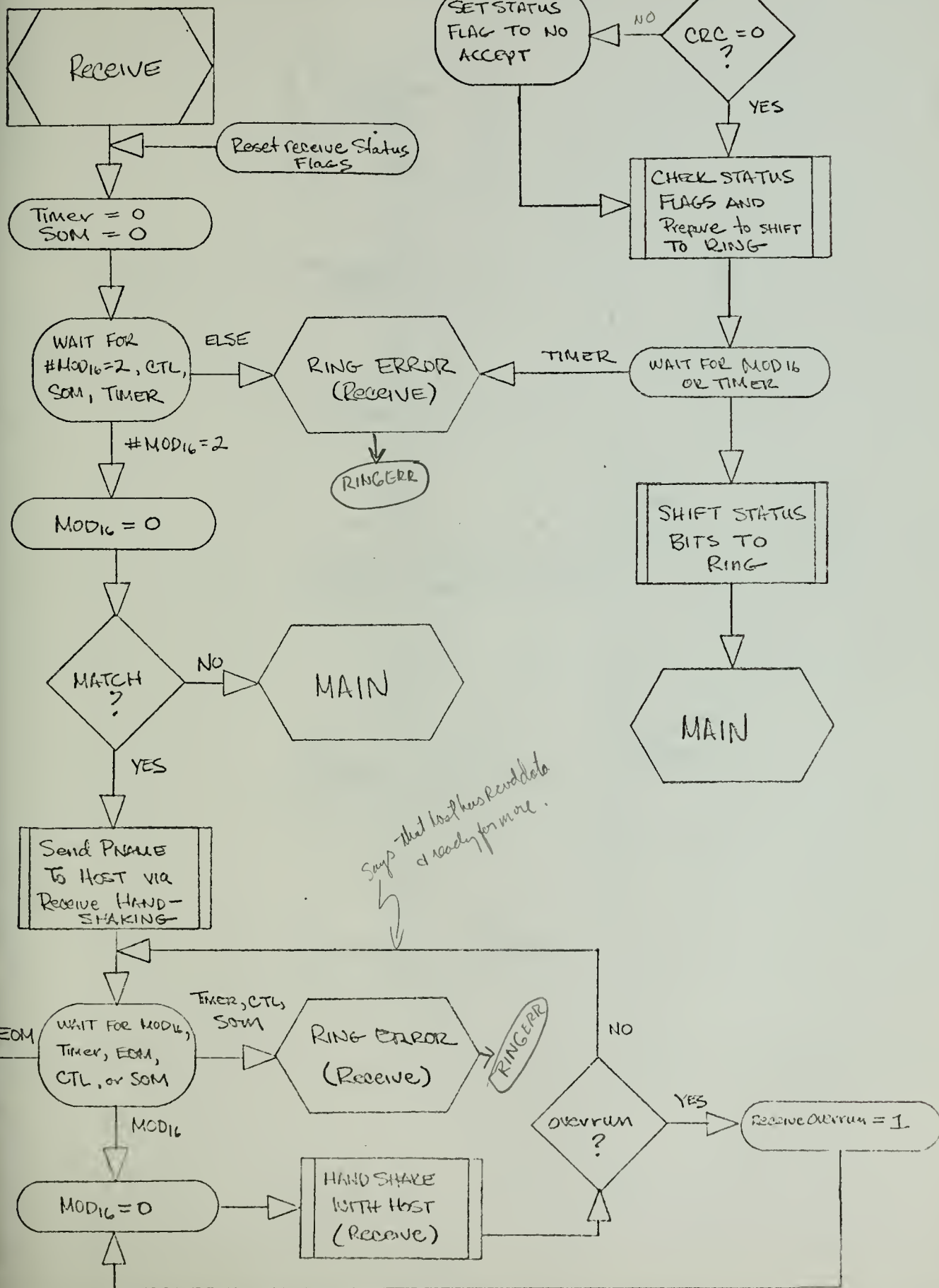
When
Disconnect = 1



when SOM = 1

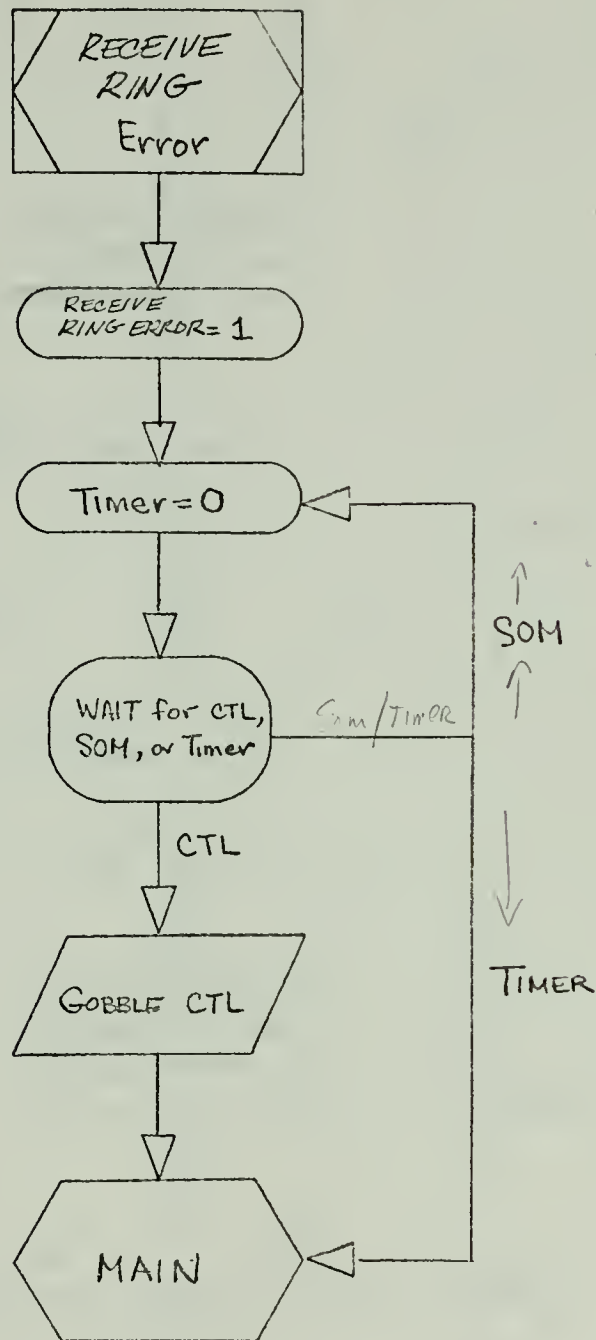
RECEIVE PROCEDURE

at EOM Receipt



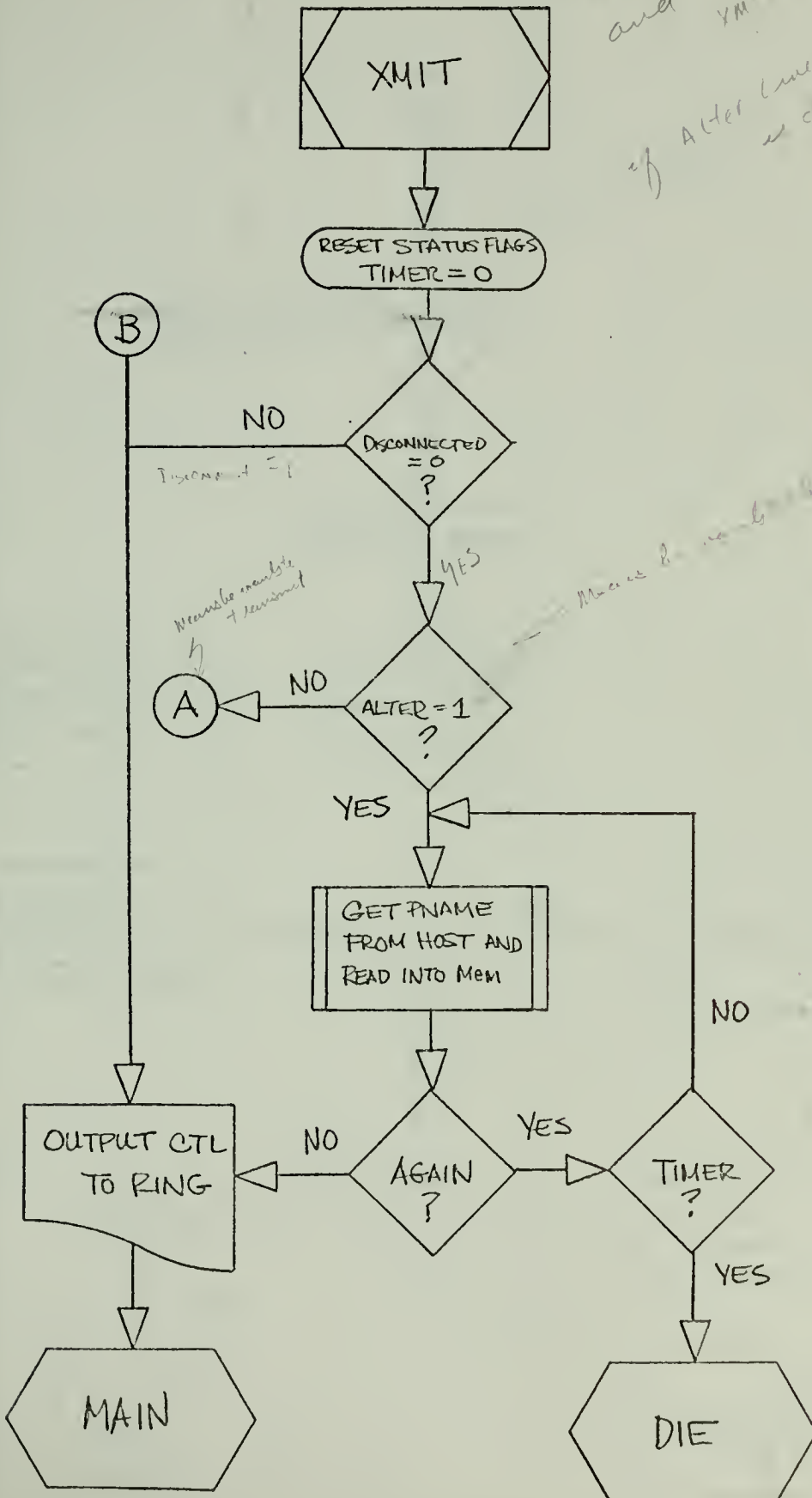
R RINGERR
RECEIVE RING ERROR PROCEDURE

Timer calc = 30m Timeout, wcy

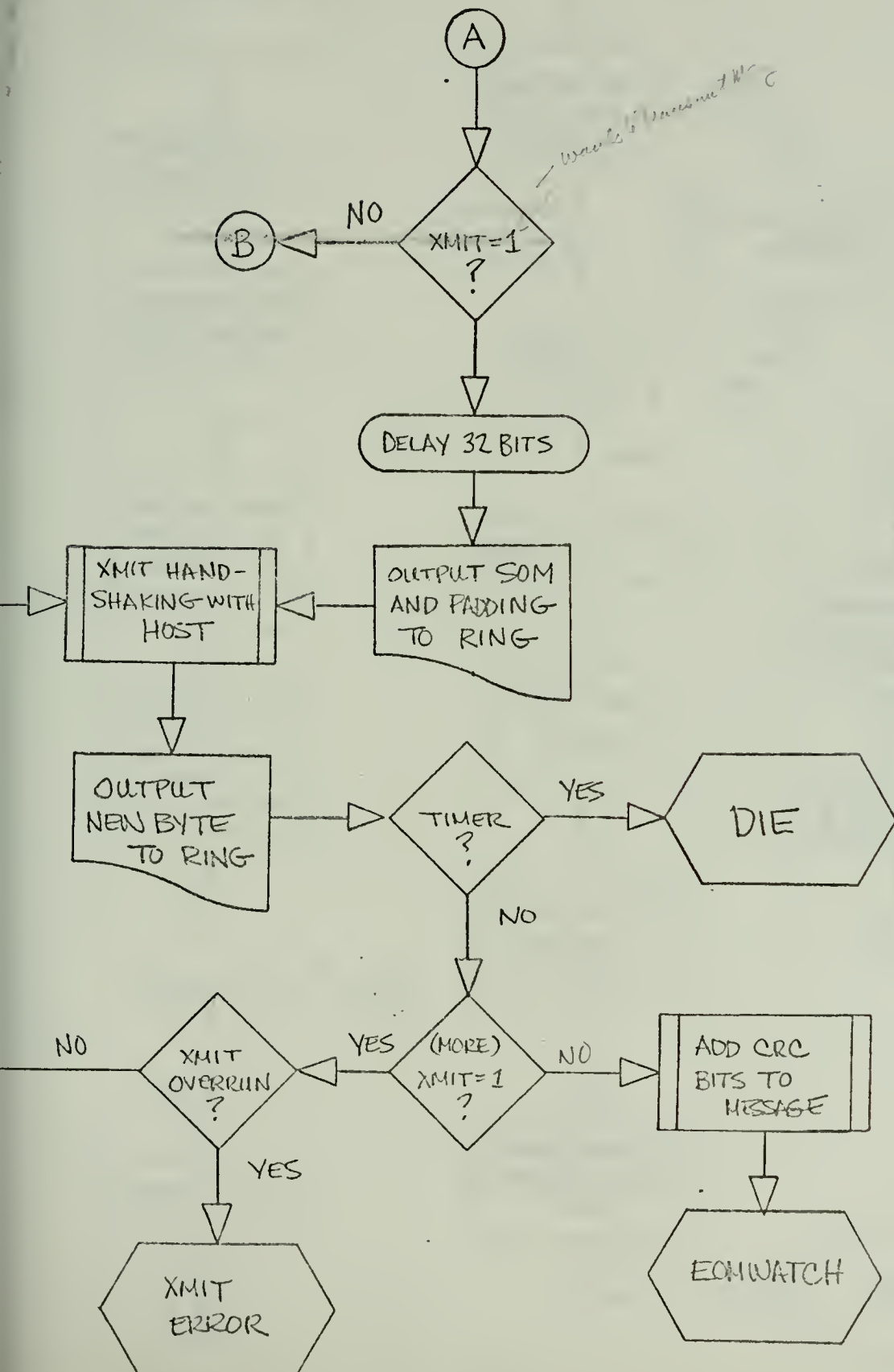


XMIT PROCEDURE

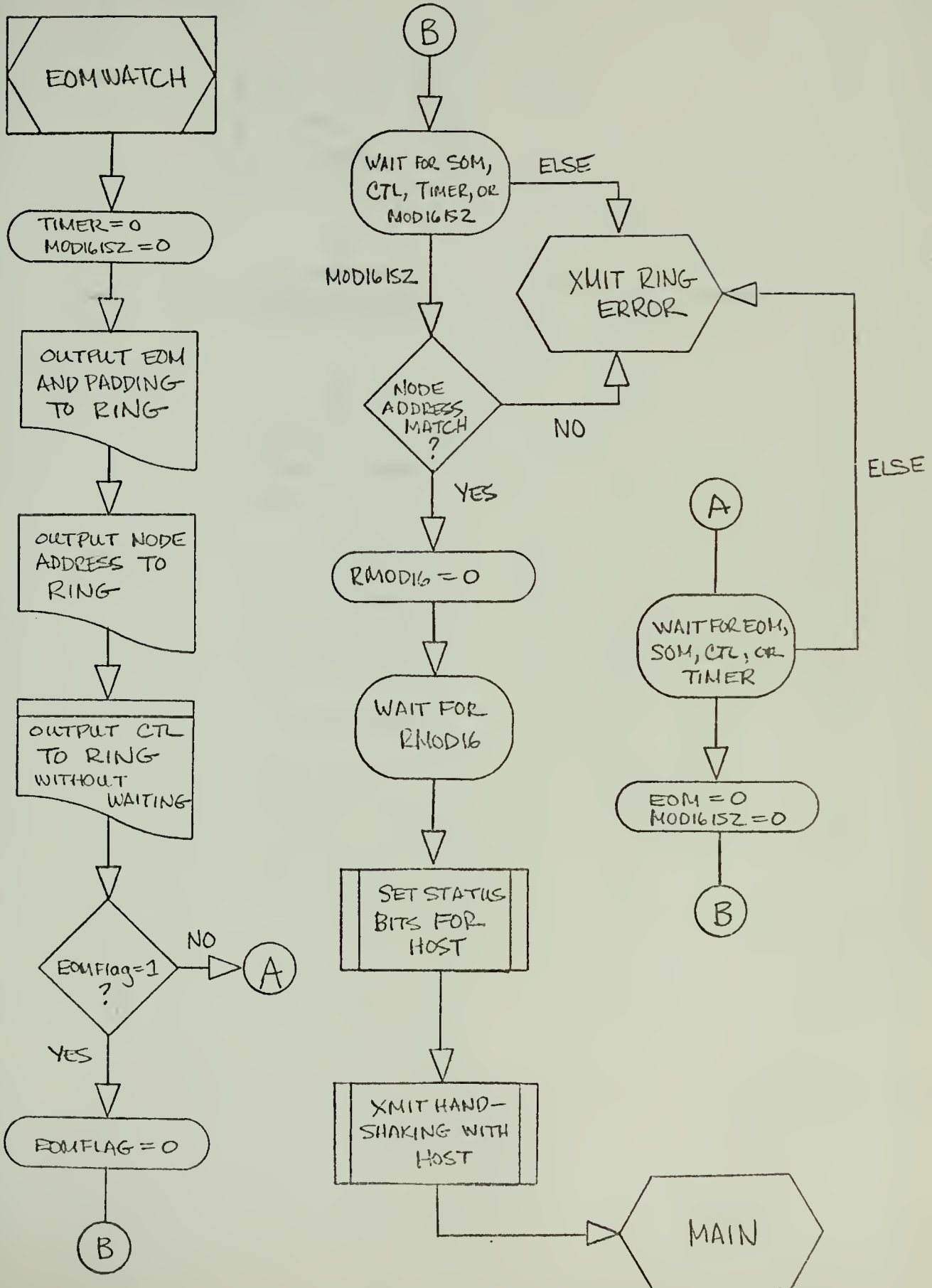
Have Read with
and request is present
XMIT line
if Alter line has reg. then name user
is changed!



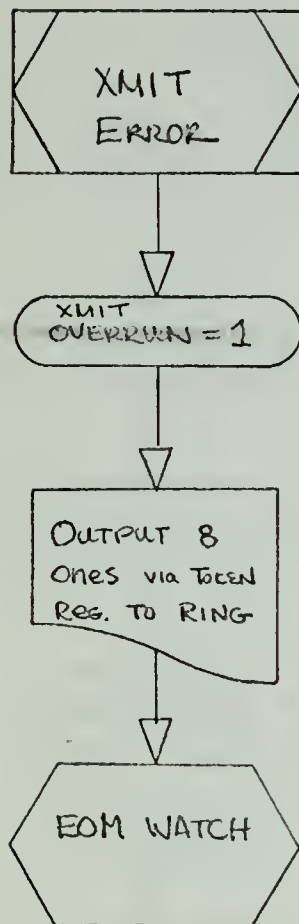
XMIT PROCEDURE (cont)



EOMWATCH PROCEDURE

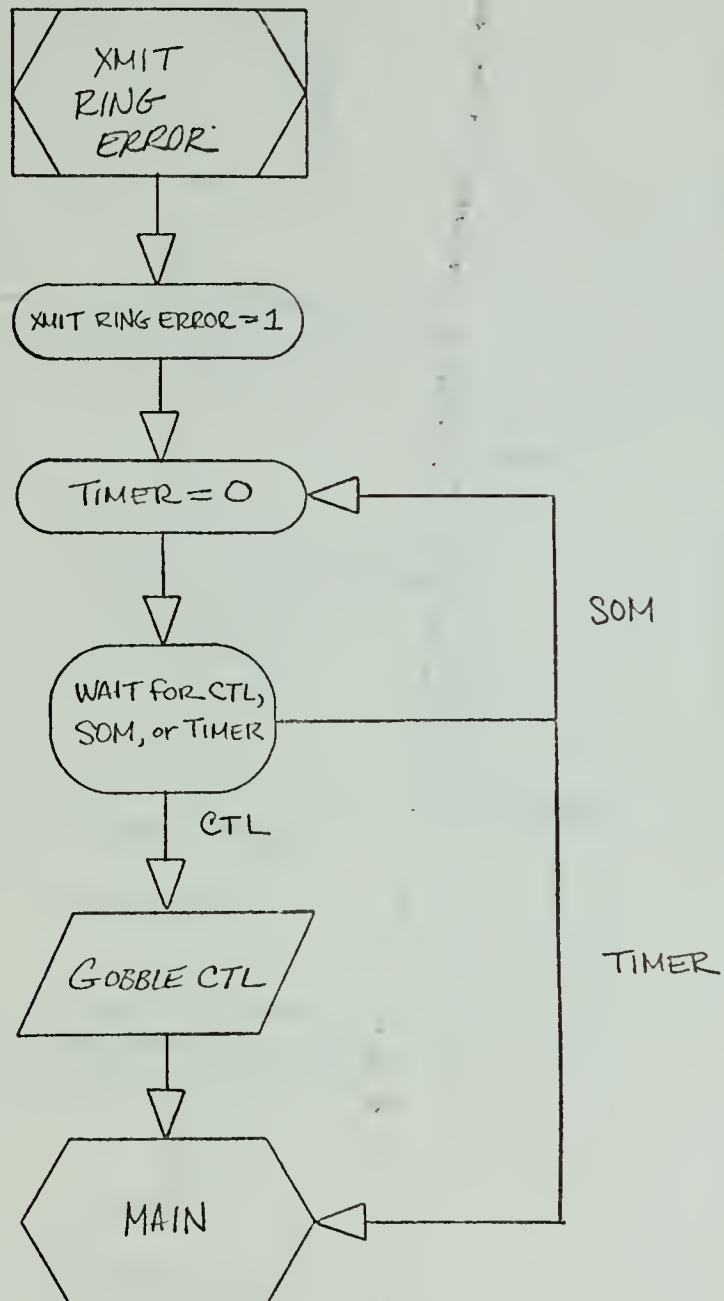


XMIT ERROR PROCEDURE

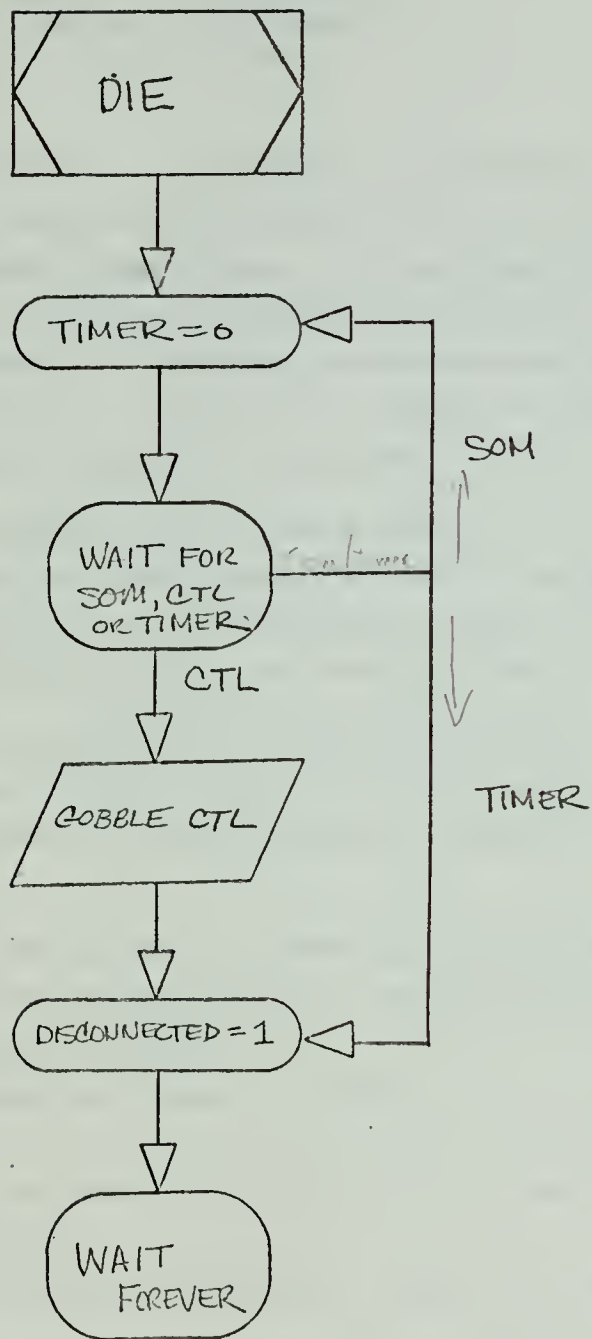


X R I N G E R R

XMIT RING ERROR PROCEDURE



DIE PROCEDURE



Appendix 2

SMAL: A Symbolic Microcontroller Assembly Language

Gary A. Kildall
Computer Science Group
Naval Postgraduate School
Monterey, California
April, 1974

I. Introduction

A simple microcontroller has been designed by the Computer Science Group at the Naval Postgraduate School (Brubaker [1]) which can be used to replace many IC's in random logic designs. The microcontroller is intended to be the heart of a particular design, with additional random logic modules at the periphery, as required. The microcontroller performs only simple tests and operations, with no ALU or subroutine mechanisms (these mechanisms are added externally, if required).

Although the microcontroller is discussed in detail in Reference [1], it is briefly reviewed here for completeness. Basically, the microcontroller has 32 "input ports" and 32 "output ports," where each port is a single bit line to external modules, as shown in Figure 1.

An 8-bit data bus is also provided for passing information to external modules. An 8-bit register is also provided for controlling program flow externally. The use of these ports and registers are described in detail in Section V.

The microcontroller instructions are stored in Read-Only-Memory (ROM), where the ROM is divided into 256 byte "pages." The instruction set includes the following simple functions

- (a) unconditional branch to a specific address in the range
0-32767
- (b) branch on input port true (1) or false (0) to a specific
page location (0-255)
- (c) Strobe a specific output port and place data on the data bus.

The purpose here is to describe a simple assembly language for writing programs for this microcontroller. The language, called SMAL, is written in PL/M (Intel, [2]), and runs on the Intellec-8 or Intellec-80 developmental system (Intel, [3]).

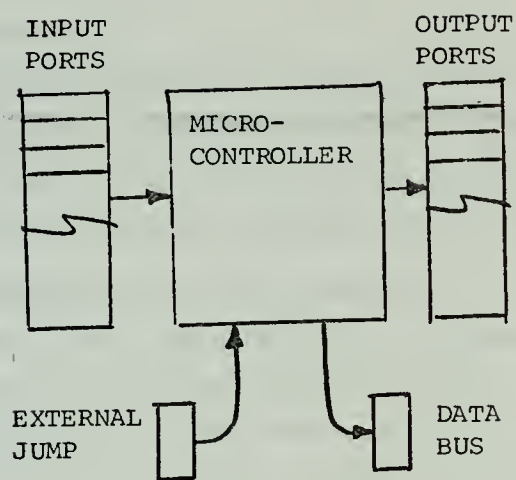


Figure 1. Microcontroller organization.

It is assumed that the reader has a basic familiarity with the micro-controller architecture throughout the discussion which follows. Further, it is assumed the reader is familiar with the Backus notation used to describe language syntax, although the examples used should suffice to present language forms.

II. The SMAL System

As mentioned previously, the SMAL assembler executes on an Intel developmental system. The machine code for SMAL is in the standard hexadecimal format (Intel, [4]), and is loaded with the standard Intel monitor. The SMAL assembler requires approximately 3K of program memory.

The assembler runs in three passes. The first pass performs the label resolution, while the remaining two passes generate Intel hexadecimal tapes for PROM or ROM programming. Two passes are required for tape generation since the microcontroller word size is 16-bits, thus requiring two 8-bit words in parallel for each memory location. The high order bytes are punched on pass-2 and the low order bytes are punched on pass-3.

III. Operating Procedures

After loading the SMAL assembler into the memory of the Intellec, the monitor command

G10₂

is issued to transfer control to the first instruction of the assembler. The assembler responds with

#000

indicating that it is ready to accept the first SMAL statement, beginning at location 000_{16} in the microcontroller memory. As instructions are typed, the code address is incremented. At any given time, the value

#hhh

at the start of a line indicates the location where the next instruction is inserted.

Since the assembler requires two more passes on the source program, the user may wish to have the paper tape punch "on" so that subsequent passes can be read through the tape reader. In this case, the line numbers are also punched on the paper tape, but are ignored on subsequent passes, or if the tape is re-run after correction.

The end of the assembly is denoted by the symbols

\$\$

The assembler will immediately punch a leader of 40 "nulls" and begin the second pass. The source program is re-read, and the high order bytes are punched by the assembler. Each high-order hexadecimal record is preceded by the symbol 'H'.

At the end of pass-2, the assembler again punches a leader and then starts pass-3. Again, a hexadecimal tape is produced; this time the low order bytes are punched, preceded by the symbol 'L'. The assembler halts after pass-3.

The assembler prints a symbol table at the end of the first pass if the assembly is terminated with

\$S

instead of '\$\$'.

As a simple introductory example, the following program checks input port 3 unit it changes to true. On a true input condition, the value on the data bus is changed from 0 to FF₁₆, and output port 5 is strobed. The program repeats this process after input port 4 to changes to false.

```
#000 /SIMPLE EXAMPLE OF A
#000 /MICROCONTROLLER PROGRAM
#000 CHANGE=3; REPEAT=4
#000 BUS=5
#000 /CHANGE REPEAT, AND BUS ARE SYNONYMS
#000 /FOR 3, 4, AND 5, RESPECTIVELY
#000
#000 START, BUS :=0 /SET BUS TO 0
#001      -CHANGE = :* /LOOP UNTIL PORT 3 IS TRUE
#002      BUS :=OFFH /SET BUS TO HEX FF
#003 LOOK, REPEAT =:START /LOOP WHEN 4 IS 0
#004 =: LOOK /OTHERWISE REPEAT THE PROGRAM
#005 $$
```


In general, the symbol '=' is used to assign assembly time values to a symbol, the character ',' is used as a label delimiter, the symbols '=: ' and ':=' are used in conditional and unconditional branches, and in port assignments, while the minus symbol '-' denotes a false conditional test and the symbol '=:' denotes an external jump. Note that comments begin with a '/' and end at the next carriage return symbol (denoted here by '2'). Multiple statements can be placed on a single line with the symbol ';' separating them. In all cases, the ';' is equivalent to a carriage-return. The exact language details are given in sections which follows.

IV. Error Messages

Errors in the assembly language source are flagged with the symbol '%' followed by a bell and a single error character. Note that although these characters are punched on the paper tape if the punch is on, they will be ignored by subsequent passes, or if the tape is completely re-run. The SMAL error characters are:

- S - error in statement syntax
- X - symbol table overflow*
- Ø - error in operand
- V - invalid port address
- P - off page reference in conditional jump
- D - definition error
- E - superfluous characters at end of statement
- undefined symbol (detected at end of assembly); the symbol is printed.

A simple sequential statement editor is built-in to the SMAL assembler to aid correction of paper tapes. This editor is described in detail in a later section..

*Each symbolic name requires $n+3$ symbol table locations, where n is the length of the name. The symbol table size is changed by altering the value of "symsize" in the SMAL assembler source program. This value is initially set at 200 bytes. There is no restriction on program length.

V. The SMAL Language

The basic tokens of SMAL are discussed first, followed by the syntax for the individual statements. In each case, the syntax is specified using BNF (Backus-Naur Form), the semantic actions are specified informally in English, and examples are given in each case.

A. The tokens of SMAL are similar to those of PL/M for

<identifier> and <number>

That is, an <identifier> is a sequence of up to 32 letters and digits, where the leading character is a letter. A <number> is an integer value in the range 0 to $2^{16}-1$, specified in one of the following bases:

<u>base</u>	<u>base indicator</u>	<u>valid digits</u>
binary	B	0,1
octal	Ø or Q	0,...,7
decimal	D or unspecified	0,...,9
hexadecimal	H	0,...,9,A,B,C,D,E,F

A <number> is a sequence of digits, followed by the base indicator. The leading digit must always be a decimal digit (0 will always suffice for hexadecimal numbers), and must be valid digits for the selected base.

examples

valid <identifier>s are

X INPUT BUS REPEAT
X2 X2Y3 LONGSYMBOLNAME

invalid <identifier>s are

3X (leading symbol not a letter)
X\$Y (contains a character which is not a letter or a digit)
REALLYLONGSTRINGOFSYMBOLSUSEDFORSYMBOLICNAME
 (symbolic name is too long)

valid <number>s are:

1 1D 123 80H 0F3H 25Q
11011B 3F5DH 772Ø 772Q OFFH

invalid <number>s are

65539 (number exceeds 65535)
FFH (hexadecimal number requires a leading decimal digit)
823Q (invalid digits used in an octal number)

B. The syntax of a <program> in SMAL is now given.

syntax

- ```

1.1. <program> ::= <statement set> <eof>
1.2. <statement set> ::= <statement> | <statement set> <sep> <statement>
1.3. <statement> ::= <label field> <basic statement> <comment>
1.4. <eof> ::= $$ | $S
1.5. <sep> ::= ; | ,

```

semantics

A program is a sequence of <statement>s separated by carriage-returns or semicolons, where the last statement is followed by double dollar signs, or a dollar sign followed by an S. In the latter case, the value of each symbol used in the program is printed.

Although not reflected in the syntax, a control-I (denoted by  $\uparrow$ I) character can be used between the statement elements to "tab" to position across the line. The tab positions are defined as 1, 8, 15, ...,  $7n+1...$  (i.e., every seven columnS) across the teletype line. The use of tabs generally reduces the paper tape length since one character is used to represent several blanks.

All statements are "free-field", and thus are not dependent upon particular columns of the teletype line. Note also that Rubout and Line-feed characters are always ignored on input. Thus, paper tapes can be prepared "off-line" for later assembly.

C. The syntax of the statement elements is given below.

syntax

- ```

2.1. <label field> ::= <label element> | <label field> <label element>
2.2. <label element> ::= <identifier>, |
                                <number>, |
                                <empty>
2.3. <base statement> ::= <value definition> |
                            <unconditional jump> |
                            <conditional jump> |
                            <output> |
                            <external jump> |
                            <empty> | <literal>

```


- 2.4. <comment> ::= /<comment string>|<empty>
- 2.5. <comment string> ::= {a sequence of arbitrary characters,
not including ';' '2', '#', or '%'}
2
- 2.6. <empty> ::= {the null string of symbols}

semantics

The <label field> is a sequence of zero or more <label elements>, where each <label element> is an identifier or a number. The labels are separated from one another, and from the statement being labelled by the ',' symbol.

If the label is a <number>, then the origin of code generation is set to this value. If multiple <number>s are encountered in a <label field>, code generation begins at the rightmost such value. The value of a numeric label must be in the range 0 to 32767. Note also that the code origin may be set to an area where code was previously generated. In this case, additional output machine code records are produced for this area of memory.

If the label is an <identifier> then two cases must be considered. If the <identifier> has not previously occurred, then the <identifier> takes the value of the current code location (and is subsequently completely synonymous with this value). If the <identifier> has occurred previously as a label, or as a defined identifier (see <value definition> below), then the <identifier> already has an associated value. This value is then used in the same manner as a <number> to re-originate code generation at a (possibly) different location.

examples

100H, code generation begins at $100_{16} = 256_{10}$
 START,10H, assuming the location counter is zero upon entry,
 and START has not previously occurred, START takes
 the value 0, and code generation begins at $10_{16} = 16_{10}$.

Again, there are no column dependencies in the <label field>. All labels are identified by the comma which follows. Further, note that the <label field> may be omitted altogether, in which case code generation continues at the next sequential location.

A <comment> can appear following the <basic statement>, and continues to the next semi-colon or carriage-return. All symbols in a <comment> are read but ignored by the assembler. Since a <basic statement> is optionally <empty>, it is possible to write a <comment> as the only entry in the <statement>.

D. The syntax of the <basic statement>s is now presented.

syntax

- 3.1. <value definition> ::= <identifier> = <right part>
- 3.2. <unconditional jump> ::= =: <right part>
- 3.3. <conditional jump> ::= <port reference> =: <right part>
- 3.4. <port reference> ::= <port value> | -<port value>
- 3.5. <output> ::= <port value> := <right part>
- 3.6. <external jump> ::= <external reference> =:: <right part>
- 3.7. <right part> ::= *|<number>|<identifier>
- 3.8. <port value> ::= <number>|<identifier>
- 3.9. <literal> ::= <number>|-<number>|<identifier>|-<identifier>
- 3.10. <external reference> ::= <number>|<identifier>

semantics

A <value definition> is used to associate a particular number with an <identifier> name. The <identifier> must not appear elsewhere on the left of a <value definition>, nor can it occur previous to this statement as a statement label. If these rules are observed then the <identifier> defined by the <value definition> can be used in place of the numeric result of the <right part>.

The <right part> can be one of three types. If it takes the form '*' then the numeric value of the <right part> is the current code location (after all elements of the <label field> are processed). If the <right part> is a <number>, then the value is simply the number itself, which must be in the range 0 to 32767. If the <right part> is an <identifier> then the value of the <right part> is the value of the <identifier>. That is, the <identifier> must appear elsewhere (before or after) as the left part of a <value definition>, or as a statement label. In this case, the value of the <identifier> is treated in exactly the same manner as a <number>.

examples

X = 55Q

Y = Z1 (Z1 defined elsewhere)

GAMMA = *

semantics

The <unconditional statement> causes microprocessor program control to transfer to the absolute memory location given by the <right part>. As above, the <right part> must evaluate to a numeric value in the range 0 to 32767.

examples

=: 500H

=: X (X defined elsewhere)

=: * (infinite loop)

semantics

A <conditional jump> is used to conditionally alter program control to a location within the page containing the jump instruction. The value of <port reference> is either a <number> or an <identifier> which evaluates to a number through a <value definition> or labeled statement. The resulting <port value>, however, must always evaluate to a number p in the range 0 through 31. If the <port value> is preceded by a minus sign, then the jump takes place on a 0 value on input line p, otherwise the jump is taken on a 1 value at port p. Program control continues to the next memory location if the condition is not met.

The jump location which is used when the condition is met comes from the value of the <right part>. As above, the <right part> must evaluate to a number k in the range $0 \leq k \leq 32767$. Note, however, that if the value of the code location counter is c after processing all statement labels, then it must be the case that

$$\left\lfloor \frac{c}{256} \right\rfloor = \left\lfloor \frac{k}{256} \right\rfloor$$

(where $\lfloor n \rfloor$ denotes the "integer part" of n). That is, the destination of the conditional jump must be to a program location on the same page as the conditional jump instruction.

examples

5 :=: 100H (jump to 256 if input port 5 is 1)
X :=: 50 (jump to location 50 if the port given by X's
 value is 1)
-31 :=: * (jump to this instruction while port 31 is 0)
-GAMMA :=: DELTA (jump to the location given by DELTA's value if
 the input port given by GAMMA's value is 0).

semantics

The <output> statement probes an output line and loads data on the 8-bit data bus. In this case, the <port value> is similar to the description above (i.e., it must evaluate to a number in the range 0 through 31), but instead designates a particular output line to be strobed. The <right part> must evaluate to an operand that can be placed on the data bus, and thus is restricted to the range 0 through 255.

examples

15 := 5 (place a 5 on the data bus and strobe output line 15)
X := OFFH (place FF₁₆ on the data bus and strobe the output
 line given by X's value)
XYZ := VAL (place VAL's value on the data bus, and strobe the
 line given by XYZ's value).

semantics

In the <external jump> command, which takes the form X ::= Y, an unconditional jump to location Y occurs with the exception that the low order bits of Y are replaced by bits from external source X. From 0 to 8 bits of Y may be replaced. The number and source of the external bits is a function of address multiplexing circuitry added to the micro-controller for particular applications.

In general, the jump external command can be used as an externally selected "case statement." For example, the jump external operation can be used to rapidly interpret an encoded command (e.g. an op code) received from external hardware. Y specifies the base address of a table and the external bits specify the entry into the table. Each entry into the table normally contains an unconditional jump to a routine to handle the particular command represented. Note that the placement of such a jump table is critical. For example, if 4 bits are being replaced, the table

must be located on a word address that is a multiple of 16.

examples

```
X ::= Y          (rightmost two bits to be obtained from external
4,Y, =: Y1       source X)
=: Y2
=: Y3
=: Y4
```

semantics

The <literal> statement allows the programmer to place literal constants into the program storage area. The form <number> evaluates to a constant in the range 0 to 65535. If the <literal> is an <identifier> then the identifier name must appear elsewhere in a <value definition> or as a statement label. In this case, the literal becomes the value associated with the <identifier>. If -<number> or -<identifier> is used, then the value v resulting from the <number> or <identifier> is "inverted." That is, the value which is taken is

$$65535 - v$$

Note also that the microcontroller inverts the rightmost five bits of a memory word when it is fetched from memory (Brubaker [1]), and thus the rightmost five bits of the literal are always stored in inverted form in the ROM so that they will come from memory in "true" form when they are eventually fetched.

examples

```
5
X
-OFE32H
-XYZ
```

Editing Commands

In order to simplify the process of correcting source tapes, a tape editor is included in the SMAL system. All editing commands are entered in the "blind" mode which prevents them from appearing on the output tape. The available commands are:


```

(ctl)L    -- generates a tape leader of 40 nulls
(ctl)A    -- assemble the remainder of the program
(ctl)A#hhh -- assemble down to line #hhh
(ctl)Annn -- assemble nnn lines of source code
(ctl)S#hhh -- skip down to line #hhh in the source tape
(ctl)Snnn -- skip nnn lines of code on the source tape
(ctl)P    -- print toggle, turns the printing of the program on and
            off during assembly

```

NOTE: (ctl) represents the control key and must be typed at the same time as the first symbol in the command.

For example, if a source tape with an error in line #5E is to be corrected, the following commands would be applicable:

```

(ctl)L,
(ctl)A#5E,
(ctl)S1,
<type correct statement>
(ctl)A,

```

VI. References

1. Brubaker, R. H. A general purpose microcontroller, Computer Science Group, Internal Document (see enclosure) Naval Postgraduate School, Monterey, California, March, 1974.
2. A Guide to PL/M Programming, Intel Corporation, 3065 Bowers Ave., Santa Clara, California, September, 1973.
3. 8008 8 Bit Parallel Central Processor Unit, Users Manual, Intel Corporation, November, 1973.
4. Intellec 8 Microcomputer System Operator's Manual, Intel Corporation, November, 1973.

Appendix 3

A GENERAL PURPOSE MICROCONTROLLER

Raymond H. Brubaker, Jr.
Assistant Professor of Computer Science
Naval Postgraduate School
Monterey, California

March, 1974

Background

This paper describes a simple programmable control unit or "microcontroller." It was designed to provide the necessary sequence of control signals for many digital applications including (1) the interface between a "floppy disk" and a small computer, (2) a control unit for the IBM System/360 multiplexor channel, and (3) the serial telecommunications interface for the NPS Ring Network (the Ring Interface).

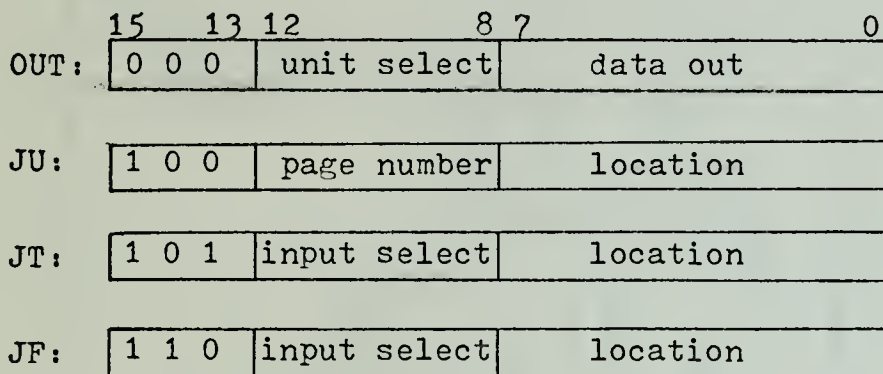
The applications of programmable controllers are almost limitless. They have become a cost-effective solution to digital control with the advent of low-cost semiconductor read-only-memory (ROM). The advantages of the microprogrammed approach are summarized below:

1. Structured designs. A more structured overall design can be achieved as random logic is reduced.
2. Adaptability. A given design can be easily changed to meet varying external needs. (Build one disk controller, and change the program to suit different computers.)
3. Debugging and update. Changes are made by altering the control store rather than rewiring.
4. Faster implementation. Designs go from conception to prototype faster with a standard, programmable control unit.
5. Fault diagnosis. Diagnostic aids can be programmed into the controller itself.

A General Purpose Microcontroller

The microcontroller described here functions basically like a small computer with a 1.1 microsecond instruction cycle and at least 256 words of reprogrammable ROM for program storage. Only four different operation codes are used: Jump Unconditionally (JU), Jump on True input (JT), Jump on False input (JF), and Output (OUT). JU causes an unconditional jump to any location on any of the 256 word pages of memory. JT tests one of 32 inputs to the controller and jumps to the specified location on the current page if the test is true; otherwise the next sequential instruction is executed. JF is similar with the jump occurring when the selected input is false. OUT briefly (100 nanoseconds, nominal) strobes one of 32 control lines and displays an 8-bit data word concurrently.

In summary, this four-instruction computer can generate a sequence of control signals, with or without data, to operate 32 distinct "devices." This sequence can be altered or repeated using jumping instructions which may be conditioned by the state of up to 32 input variables. The detailed instruction formats are shown in figure 1.



(Note that bits 0 through 12 are stored in complement form.)

Figure 1. Instruction Format

Machine Architecture

The architecture of the controller is shown in block form in figure 2. For purposes of discussion, it can be divided into four basic units: memory, instruction counter, input selector (multiplexer), and output selector (decoder). A schematic is attached to this paper.

Memory.

Instruction memory is provided by pages of 16-bit words with 256 words per page. Up to 32 pages may be attached although it appears that many complex applications can be handled with one or two-page controllers. The upper three bits of a ROM word feed a decoder to yield eight distinct opcode lines (only four are currently used). The next five bits assume a different selection role depending on the operation (see figure 1). The lower eight bits provide the address in jumping instructions and the parallel data for output operations.

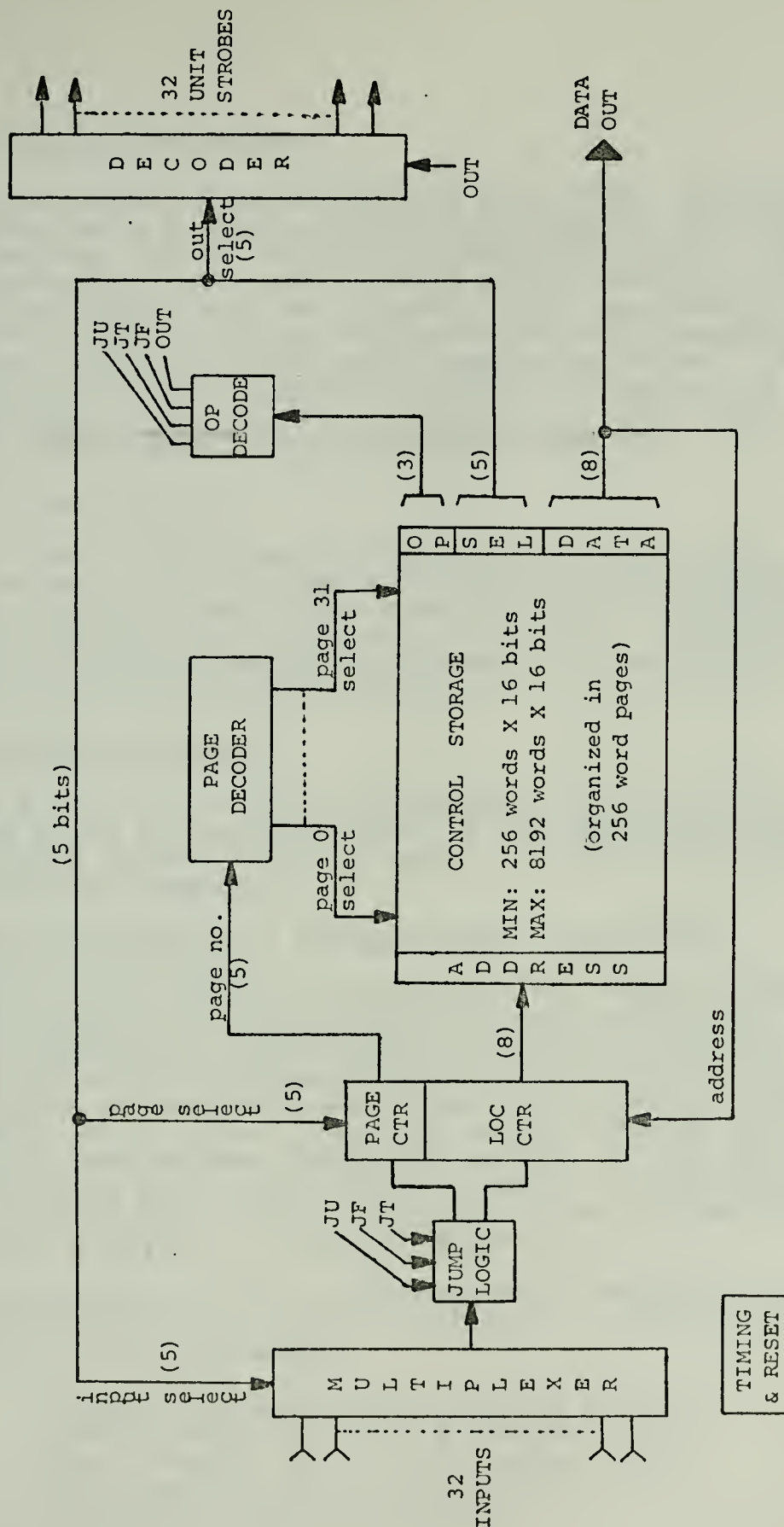


Figure 2. Microcontroller Architecture

The Instruction Counter.

The instruction counter consists of an 8-bit location counter and a 5-bit (maximum) page counter. After OUT operations, the IC is simply incremented by one. During JU operations, the lower 13 bits of the instruction are parallel-loaded into the counter. On a successful JT or JF operation, 8 bits are loaded into the location counter while the page number remains unchanged. Note that it is not possible to jump out of a page using a conditional jump, but it is possible to "fall" off a page during normal incrementation of the IC.

The Input Selector.

Bits 8 to 12 of JT and JF instructions are used to select one of the 32 inputs for testing. This selected value, coming from a 32-to-1 multiplexor, is fed into the branching logic along with the op-code. Together they are used to control the loading and incrementing of the page and location counters.

The Output Selector.

Bits 8 to 12 of an OUT instruction select one of 32 output lines using a 5-to-32 decoder. A 100 nanosecond pulse is placed on that line just prior to selecting the next instruction from ROM.

Sample Application: A Traffic Signal Controller

Consider the problem of controlling the traffic signals in a typical 4-way intersection. Let's assume that North-South (NS) is the favored direction, that is, the NS light will stay green unless the East-West (EW) walk button is pressed or a car drives over a sensor buried beneath the EW lanes. Just for variety (and to make the control problem more difficult) we will set the NS lights to flashing yellow, and the EW lights to flashing red during late night hours. Figure 3 presents a possible control sequence for such an intersection.

Implementation with the microcontroller requires that we first define the input/output characteristics of the devices to be controlled:

Traffic lights: these will have a 2-bit binary color input and a "change" input. When "change" is 1, a new "color" value is accepted and displayed. (00-off, 01-green, 10-yellow, 11-red)

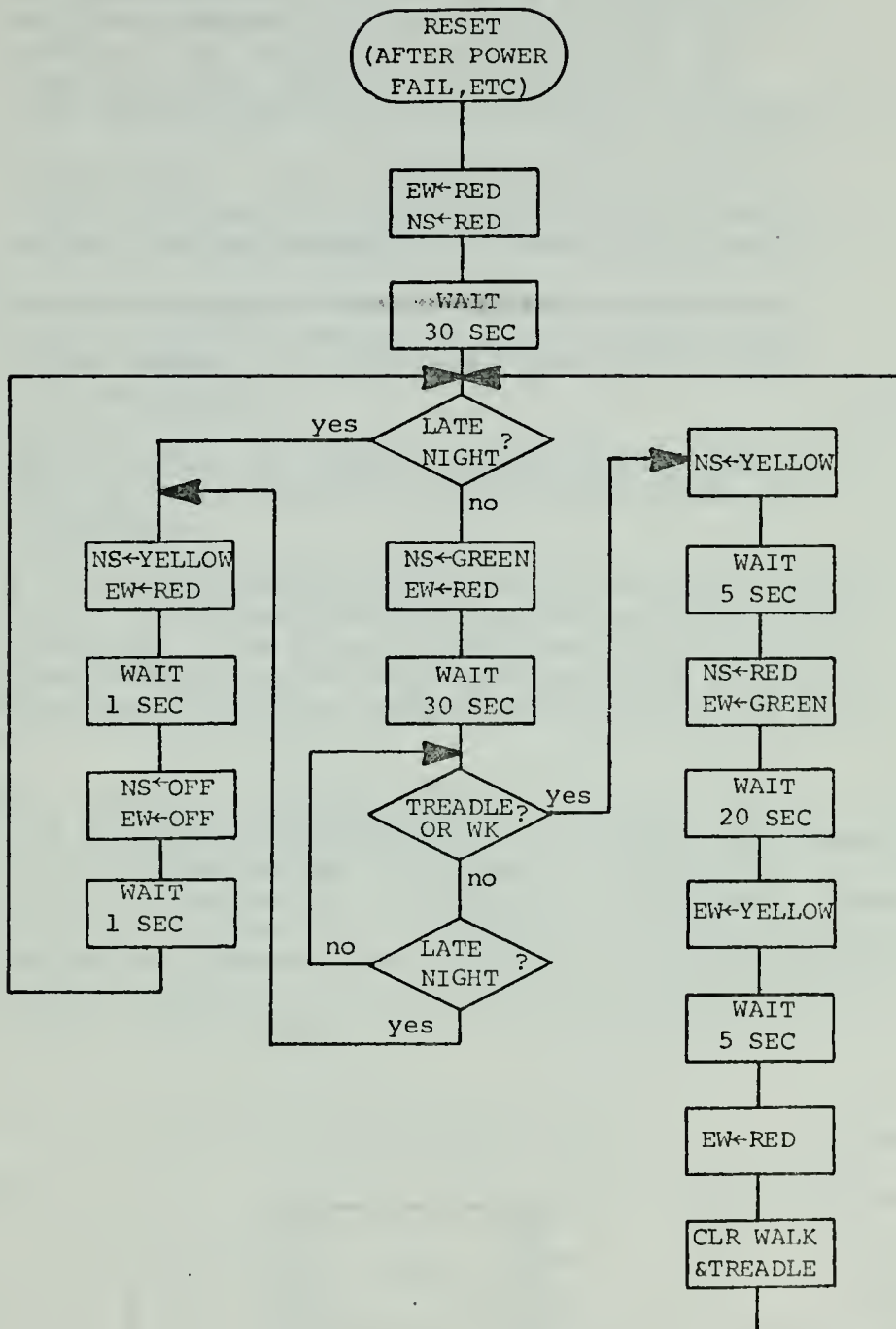


Figure 3. Control Sequence for Traffic Controller

Timer: an 8-bit binary counter which can be loaded to any value up to 255 and then self-decrements to zero at one count per second. An "expired" latch records the fact that the counter has reached zero. The latch resets when the counter is loaded.

Late night: a real-time clock which keeps track of time-of-day and provides a true output during preselected late-night hours.

Treadle: Any of a group of sensors beneath the EW traffic lanes. A 1 is latched when a car passes over and held until reset by the controller.

Walk button: As above, but indicates the request of a pedestrian to cross in the EW direction.

Figure 4 shows the hookup of the devices to the microcontroller. Note that four inputs are required for decision-making, five outputs (strokes) to reset and control the various units, and the data bus is used for setting the timer and selecting the color of both signals.

A symbolic program to implement the traffic signal controller with the microcontroller is given in figure 5. Symbols are defined using the "equal" symbol (=). Comments are indicated by a slash (/). Statement labels are set off by a colon (:). Fields of the instructions are separated by commas. The asterisk (*) is used to indicate the location of the current instruction for single-instruction loops.

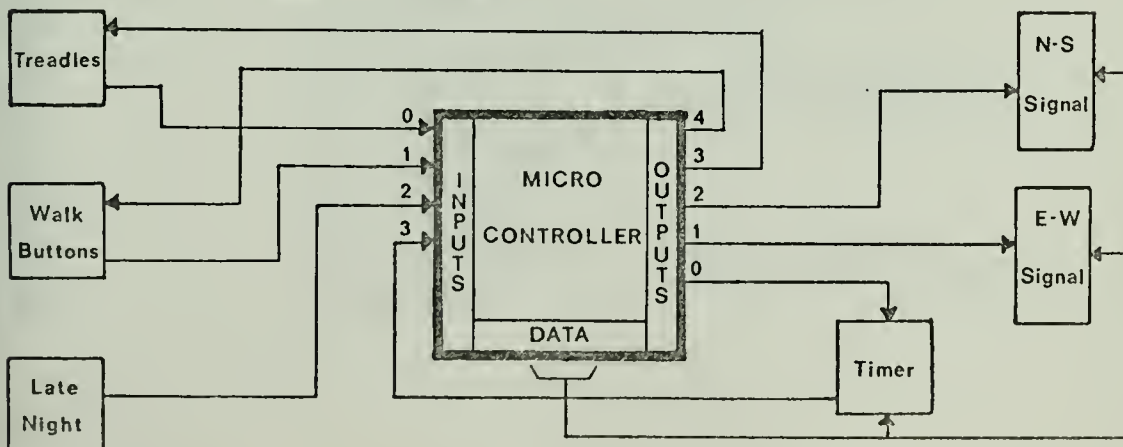


Figure 4. Traffic Controller Schematic

ADDR

STATEMENT

```

0      / INPUT DEFINITIONS
1      TREADLE=0; WALK=1; NIGHT=2; TIMEOUT=3
2
3      / OUTPUT DEFINITIONS
4      SETTIMER=0; EW=1; NS=2; CLRTREAD=3; CLRWK=4
5      OFF=0; GREEN=1; YELLOW=2; RED=3
6
7      / COME HERE AFTER POWER FAIL OR OTHER RESET
8      RESET:  OUT,NS,RED; OUT,EW,RED /BOTH LIGHTS RED
9              OUT,SETTIMER,30
10             JF,TIMEOUT,* / WAIT 30 SECONDS
11
12      /MAIN LOOP: CHECK FOR LATE NIGHT
13      MAIN:  JT,NIGHT,FLASH
14             OUT,NS,GREEN; OUT,EW,RED
15             OUT,SETTIMER,30
16             JF,TIMEOUT,* / WAIT 30 SECONDS
17
18      CHECK:  JT,TREADLE,CHANGE
19             JT,WALK,CHANGE / LOOP UNTIL EVENT
20             JF,NIGHT,CHECK /
21
22      / LATE AT NIGHT....FLASH THE LIGHTS
23      FLASH:  OUT,NS,YELLOW; OUT,EW,RED
24             OUT,SETTIMER,1
25             JF,TIMEOUT,* / WAIT ONE SECOND
26             OUT,NS,OFF; OUT,EW,OFF
27             OUT,SETTIMER,1
28             JF,TIMEOUT,* / WAIT ONE SECOND
29             JU,MAIN / GO SEE IF IT'S MORNING YET
30
31      / CHANGE EW TO GREEN IN RESPONSE TO WALK BUTTON
32      / OR TREADLE
33      CHANGE:  OUT,NS,YELLOW / SEQUENCE FROM NS GREEN
34             OUT,SETTIMER,5 / TO EW GREEN
35             JF,TIMEOUT,*
36             OUT,NS,RED; OUT,EW,GREEN
37             OUT,SETTIMER,20
38             JF,TIMEOUT,* / WAIT 20 SECONDS
39             OUT,EW,YELLOW / SEQUENCE EW TO RED
40             OUT,SETTIMER,5
41             JF,TIMEOUT,*
42             OUT,EW,RED
43             OUT,CLRTREAD,0 / RESET TREADLES
44             OUT,CLRWK,0 / RESET WALK BUTTON
45             JU,MAIN
```

Figure 5. Symbolic Traffic Controller Program

The program requires 35 words of ROM storage. The ROMs are normally programmed using binary data recorded on paper tape. These tapes could be generated either manually, converting each instruction in figure 5 to binary, or with the help of a symbolic assembler. A symbolic assembler which runs on Intel's 8008 microcomputers is available. The statements accepted by this assembler are more concise (less readable at first) to make the assembler faster and smaller, and to make the speed of a teletype more bearable. The assembler makes two passes over the source program and produces a paper tape suitable for programming 1702A ROMs.

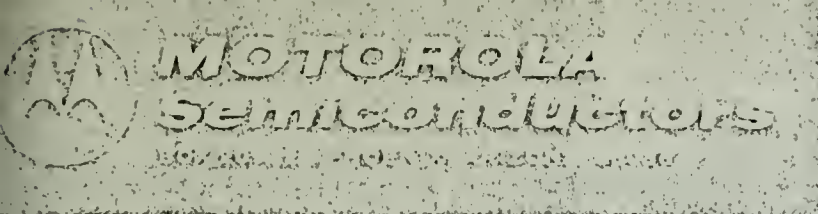
Possible Extensions

Just as in higher-level and assembler-level programming, the microprogrammer finds it necessary to repeat the same sequence of steps at several points in a control sequence. To save space in the control memory (ROM) a subroutine capability could be added. This would require the additional circuitry to stack the current IC value (CALL) and later restore a stacked IC value (RETURN). Note that this will degrade performance due to the two extra instruction cycles required to invoke the shared routine.

With 1702A ROMs the instruction cycle is limited to about 1.1 microseconds. Using newer fusible-link ROMs (or even masked ROMs in production applications) combined with high speed logic a cycle less than 200 nanoseconds is easily attainable.

APPENDIX 4

MOTOROLA BIPOLAR LSI UNIVERSAL POLYNOMIAL GENERATOR



MC8503P

UNIVERSAL POLYNOMIAL GENERATOR (UPG)

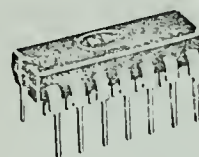
The MC8503 Universal Polynomial Generator (UPG) is used in serial digital data handing systems for error detection and correction. The serial data stream is divided by a selected polynomial and the division remainder is transmitted at the end of the data stream as a Cyclic Redundancy Check Character (CRCC). When the data is received the same calculation is performed. If there were no errors in transmission, the new remainder will be zero.

The MC8503 offers four of the more common polynomials for error detection techniques including a read forward and reverse on the CRCC-16 and CRCC-CCITT polynomial functions. These polynomials can be generated by changing the binary select codes as shown in Figure 1.

- Four Unique Polynomial Codes in One Package
- Compatible with TTL
Maximum Fan-Out = 1 TTL Load
- Data Rate = 5 MHz Typical
- Total Power Dissipation = 400 mW Typical
- +5.0-Volt Operation

BIPOLAR LSI

UNIVERSAL POLYNOMIAL GENERATOR

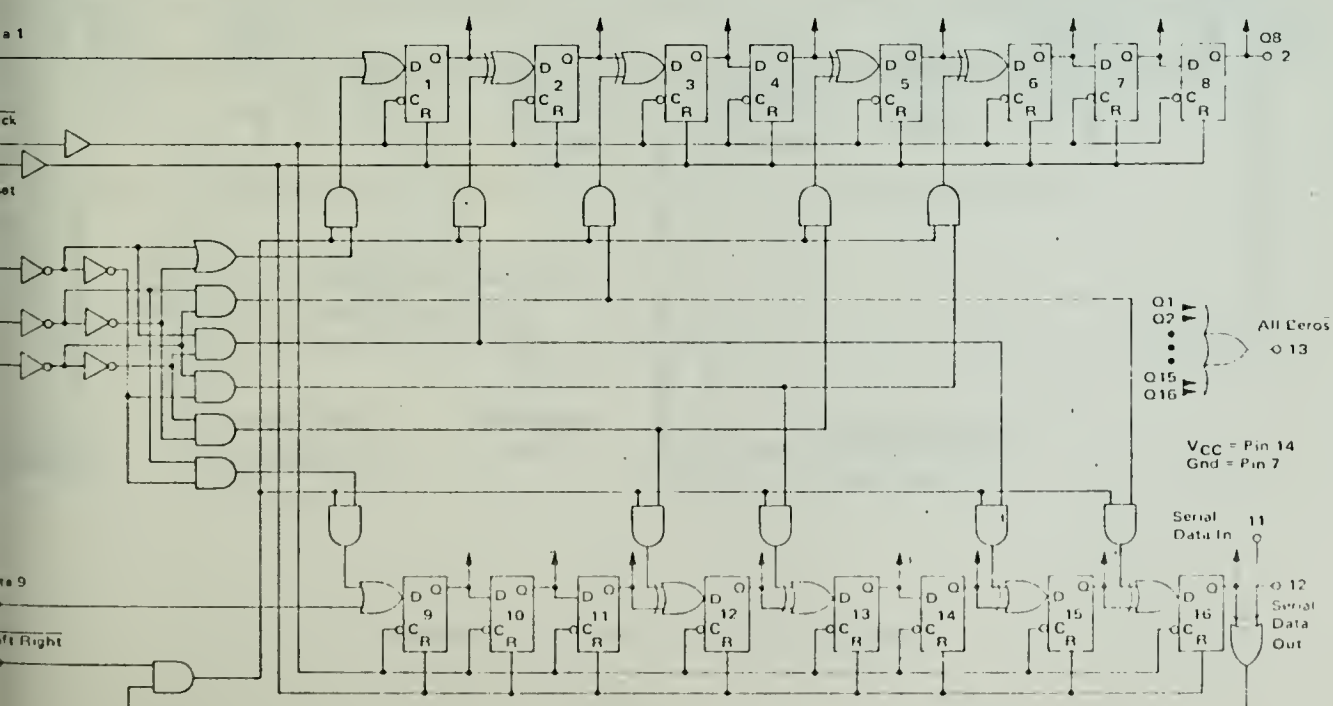


PLASTIC PACKAGE
CASE 646

FIGURE 1 — AVAILABLE POLYNOMIALS

CODE SELECT			POLYNOMIAL
X	Y	Z	
0	0	0	CRCC 16 (Fwd) $x^{16} + x^{15} + x^2 + 1$
0	0	1	CRCC 16 (Bkwd) $x^{16} + x^{14} + x + 1$
1	1	0	CRCC CCITT (Fwd) $x^{16} + x^{12} + x^5 + 1$
1	1	1	CRCC CCITT (Bkwd) $x^{16} + x^{11} + x^4 + 1$
0	1	0	LRCC 16 $x^{16} + 1$
1	0	1	LRCC 8 $x^8 + 1$

LOGIC DIAGRAM



Appendix 5

PROPOSED RING REPEATER DESIGN

Raymond H. Brubaker, Jr.
Assistant Professor of Computer Science
12 June 1974

Introduction

The Ring Interface discussed in this thesis was designed to connect to a byte parallel host on the one side and a bit serial repeater on the other. The repeater must connect directly to the inbound ring cable, receive the signal, recover clocking information, and pass on reshaped (and possibly retimed) data to the outbound cable. To design the repeater, then, one must know what type of cable is to be used, what transmission distances are required (and consider such effects as "pulse jitter"), what type of driver/receivers are to be used, and what transmission speed is to be used. The repeater was separated from the ring interface so that these questions could be deferred until the speed capability of the RI was known, and to further modularize the ring design and allow insertion of a repeater (without RI) in long cable runs.

Such a repeater is diagrammed in Figure 1.

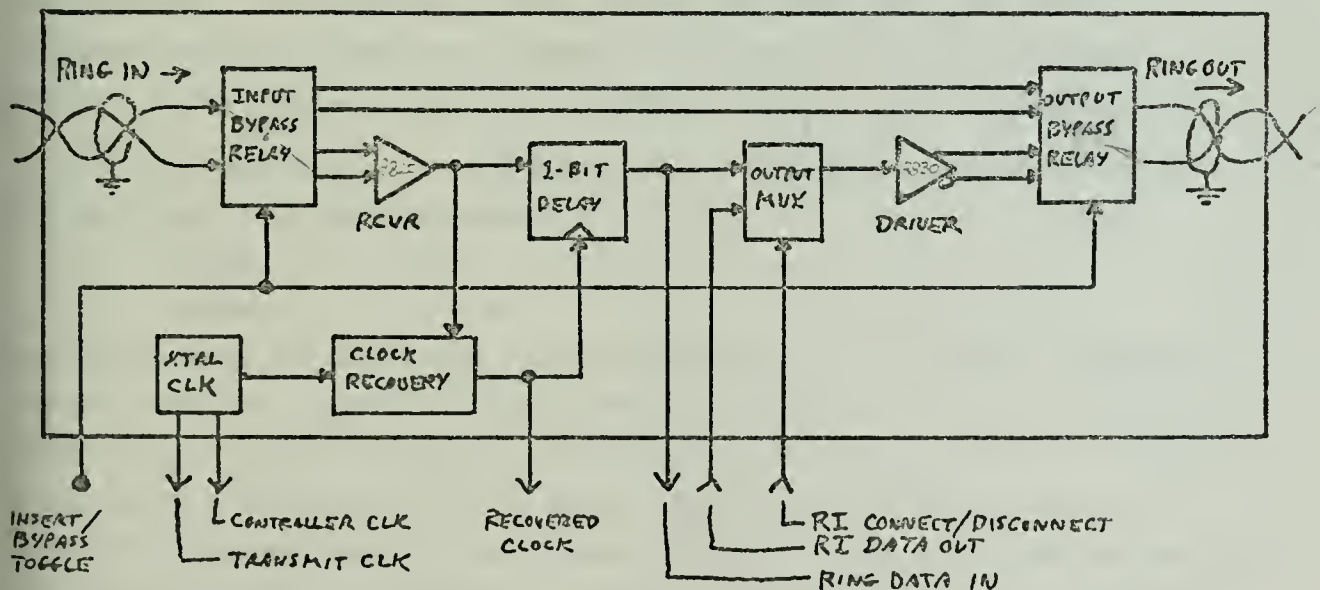


Figure 1. Repeater Block Diagram

The various components of the figure are discussed in the following paragraphs.

Ring Cable

Only a single high speed bitstream is transmitted in this self-clocking ring system. This would suggest the use of single wire, coaxial cable, or twisted pair transmission lines. Single wire lines have extreme susceptibility to coupled noise and problems with differing ground potentials. Coaxial lines have very good noise characteristics if very good grounds can be found for the shield, otherwise ground loop currents can develop and reduce noise margins considerably. The twisted pair line appears the best solution to the noise problem. It is a balanced line which can be driven differentially; in other words the voltage (or current) on one line is not of primary importance, but rather it is the difference between the voltages (or currents) which determines a one or zero bit.

The combination of twisted pair cable and differential line drivers yields a high immunity to that noise which affects each cable equally ("common mode noise"). The only problem encountered involves common mode noise which is at a very high potential with respect to ground at the receiving end. Such noise (say greater than + 15 volts) would drive most semiconductor receivers out of their operating range and cause data misinterpretation and/or destruction of the receiver. For this reason, the use of twisted pair cable with a 100% foil shield is recommended.

Shielded twisted-pair cables are available from several manufacturers. The following look very promising:

Belden	8761
Columbia	02514

Note that these are listed as "audio" grade cables. The smaller "instrumentation" grade cables such as Belden 8451 are easier to work with, but more expensive and display higher capacitance per unit length. The cable, being basically a R/C network, shows a unique rise time for a given length and type cable and particular driver characteristics. This means that as pulses get smaller and smaller (higher frequency data) they will become more and more rounded due to rise and fall times until they disappear altogether.

Adjacent bits may also interfere with each other causing a phenomenon known as "intersymbol interference" or "pulse jitter."

Thus the cable acts as a low pass filter. To avoid serious problems a rule of thumb is to restrict the data rate and/or cable runs to the point where the duration of the smallest pulse is at least four times the 90% rise time of the cable. In experiments with 1000' lengths of the cable recommended above and representative drivers, rise times on the order of 1 microsecond were measured. This would suggest frequencies giving bits of 4 microsecond duration, or 250,000 bits per second. (This would allow 125,000 bps data rate on the ring since data bits are encoded two-for-one.)

See the Fairchild reference at the end of this discussion for an excellent presentation of transmission cables, data rates, and simple measurement techniques.

Differential Line Drivers and Receivers

Integrated Circuit line drivers and receivers are available from several sources including National, Fairchild, and Signetics. The National 8830/8820 (or 7830/7820) drivers and receivers have been used successfully in experiments at NPS. The receiver (8820) accepts twisted pair inputs and provides a TTL output to interface with standard logic circuits. The driver (8830) accepts a TTL input and transmits into twisted pair cable. See National data sheets for details on these devices.

Recent advances in optoelectronics have produced optically isolated receivers with nearly total immunity to common-mode noise. Early opto-isolators were restricted to lower data rates, but recent models (see Hewlett-Packard reference) are capable of megahertz speeds and are relatively inexpensive (five dollars). Opto-isolator receivers are compatible with the National differential drivers. Such a combination should produce a virtually noise-immune system.

Bypass Relays

The purpose of the bypass relays is to simply switch the repeater "physically" out of the ring in case of power failure or during repeater maintenance. Note that switching out a repeater increases the effective

cable length between two repeaters and thus effects the cable rise time causing increased pulse jitter. This must be considered when planning cable runs and placement of repeaters.

One-Bit Delay

The one-bit delay is a single flip-flop driven at the recovered-clock rate and serves to re-time the received signal before retransmission.

Output Multiplexer

A two-to-one multiplexor is used to route data from the delay flip-flop to the ring or from the Ring Interface to the ring. The multiplexed path is controlled by the connect/disconnect line from the RI. Note that the RI is designed to "listen" to passing data, watching for a CTL token, before entering the ring (switching the multiplexer to "connect"). Thus the ring-data-in line is always valid (when the repeater is not bypassed) and is derived from the output of the delay flip-flop.

Crystal Clock

A simple crystal clock is shown in Figure 2.

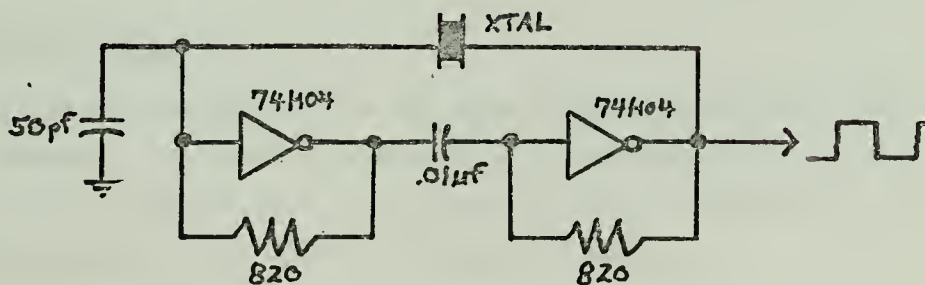


Figure 2. Simple Crystal Clock

Such a clock is quite stable and is as inexpensive as circuits using one-shot multivibrators. With proper division (using TTL flip-flops or counters) it could be used to clock the microcontroller, provide the data transmission frequency for the RI, and provide a reference for digital phase-locking in the clock recovery circuit.

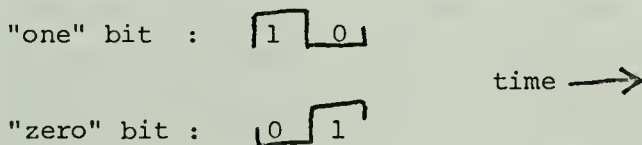
Consider three uses of an inexpensive 3.58 Mhz "TV" crystal:

- (1) For microcontroller clock: Divide-by-four gives approximately .9 Mhz or 1.1 usec per cycle which is the maximum rate for the controller.
- (2) For RI transmit clock: Divide by 16 gives 224 Khz or 4.5 usec per cycle. This is the signal for the RI to transmit a bit and corresponds to a user data rate of 112 Kbs or 9 usec per bit. (Note that this complies with the frequency limit discussed for the cable.
- (3) For recovered clock reference: See next section.

Note the important relationship between (1) and (2) above. The microcontroller can execute only eight instructions between successive user data bits (again, there are two ring bits per "user" bit). This is just enough time to check a couple of conditions and set a flag or two. The ring data rate must be decreased, or the controller speed increased (requiring faster program memory) if more processing is ever required between two adjacent bits.

Clock Recovery Circuitry

The bitstream retrieved from the ring is self-clocking in that frequent one-zero (and zero-one) transitions are guaranteed during messages. Clock pulses can be regenerated at these transitions. Data bits on the ring are sent in 2-bit packets as shown:



During receipt of tokens (CTL, SOM, EOM) up to three bit periods may pass with no transitions; the clock recovery circuitry must have sufficient "inertia" to continue with minimal frequency drift during these periods. It is also desirable to recover the clock at a frequency which is the long-term average of the incoming frequency since (1) individual incoming bits may be time distorted and (2) data will be retransmitted at the recovered frequency thus re-timing individual bits (correcting pulse jitter).

If an attempt is made to recover the clock at a long-term frequency even slightly different from the incoming rate, an overrun condition (either bit not available for transmission when needed, or bit lost when next bit arrives before previous is transmitted) is guaranteed after a finite number of bits.

The problem being presented here is one of phase-locking a local clock to an input frequency. Linear phase locking techniques may be applicable (and have been briefly considered) but seem to have a disadvantage in terms of the time required to acquire phase-lock. To explain, each node (RI) has a distinct transmit clock phase and frequency, thus two adjacent messages on the ring may require clock recovery at different phase and frequency. A linear phase locked loop, in phase with message i , could time distort message $i+1$ while trying to lock-on; subsequent repeaters could further time-distort until data is lost (probably the SOM). Some analytical modeling of this situation would be desirable.

Two paths of solution appear open. The first involves assigning phase-locking responsibility to one node (this assignment may be temporary), while the second requires investigation of digital phase-locking methods. The former would pose a threat to reliability unless timing authority migrated around the ring, but it has not been thoroughly investigated. The latter will be discussed below.

Figure 3 shows a digital phase control circuit adapted from Bennet and Davey (see references). This circuit recovers clocking information from the incoming data stream. The incoming data frequency should easily be within one percent of our local crystal generated frequency since it is also crystal controlled. Thus two adjacent bits should

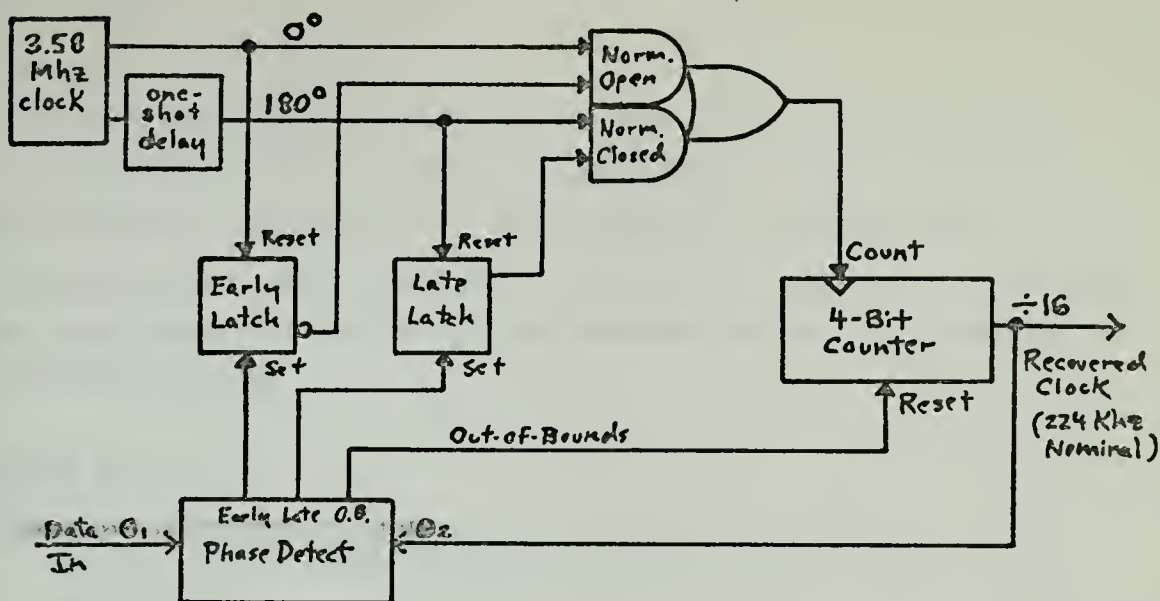


Figure 3. Digital Clock Recovery

display a phase shift of no more than 3.6 degrees (one percent of 360 degrees) with respect to the local clock. The circuit attempts to phase lock to the incoming data by deleting or adding a pulse as required to the 16-to-1 counter. This frequency change causes a phase shift of about 22 degrees (1/16 of 360). Thus the circuit maintains lock in quantum jumps of 22 degrees. Finer jumps could be attained (for a given data rate) using a faster crystal clock and higher degree of division (32 to 1, for example).

When message $i+1$ arrives, significantly out of phase with message i (and thus with the recovered clock), we would like to immediately "snap" into phase lock and track from there; that is the function of the "out of bounds" line from the phase detection circuitry.

The phase detection circuitry must compare incoming data transitions with the recovered clock and then decide to

- (1) do nothing if phases are close enough;
- (2) retard the recovered clock by setting the "early" latch if recovered pulse leads a transition;
- (3) advance the clock by setting the "late" latch if recovered clock trails a transition;

or (4) snap into phase by resetting the counter if an "out of bounds" threshold (say, 90 degrees) is exceeded.

A phase detector could be designed around one-shot multivibrators.

This mechanism should provide a clock following the long term average of the input frequency, but having the added ability to lock instantly to a new message's phase.

Suggested References

1. The TTL Applications Handbook, Fairchild Semiconductor, 464 Ellis St., Mountain View, California 94042, August 1973, Section 14.
2. Digital Integrated Circuits Data Book, National Semiconductor Corp., 2900 Semiconductor Dr., Santa Clara, California 95051.
3. Solid State Display and Optoelectronics Designer's Catalog, Hewlett Packard, 620 Page Mill Rd., Palo Alto, California 94304, July 1973, pp 23-27.
4. Bennett, W. R., and Davey, J. R., Data Transmission, McGraw-Hill, New York, 1965, Chapter 14.

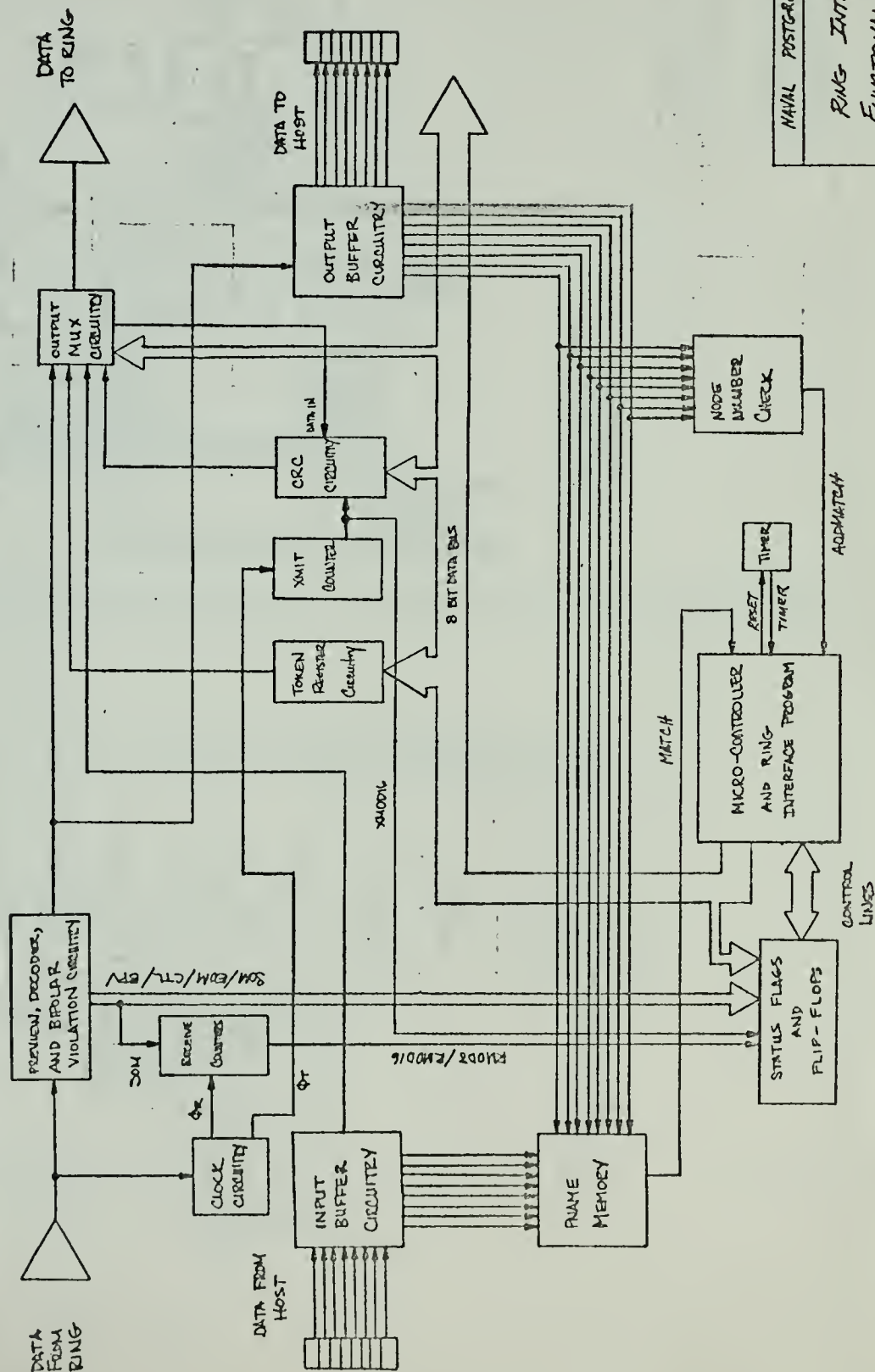
APPENDIX 6

RING INTERFACE DESIGN SCHEMATICS

In the drawings that follow, the following information is assumed. DB \emptyset through DB7 is the output data bus from the microcontroller. HI \emptyset through HI7 is the input byte from the host processor while OB \emptyset through OB7 is the output byte to the host. Data In comes from the repeater along with \mathcal{P}_T and \mathcal{P}_R , the two clocks used. Data Out correspondingly is output to the repeater.

The input/output ports I \emptyset and I1 contain the data lines between the RI and the host while I7 contains the data for the repeater. The edge connector numbers shown on the Ring Interface Circuitry Module Connections drawing are identical to the microcontroller definitions in Appendix 3 and are the input and output connections from the microcontroller to the RI circuitry.

Finally, note that DB \emptyset is used extensively in the RI design and therefore had to be amplified for this purpose. This amplification is shown in the Flag Configurations and Reset Multiplexor drawing.

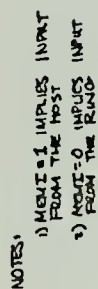


NAVAL POSTGRADUATE SCHOOL

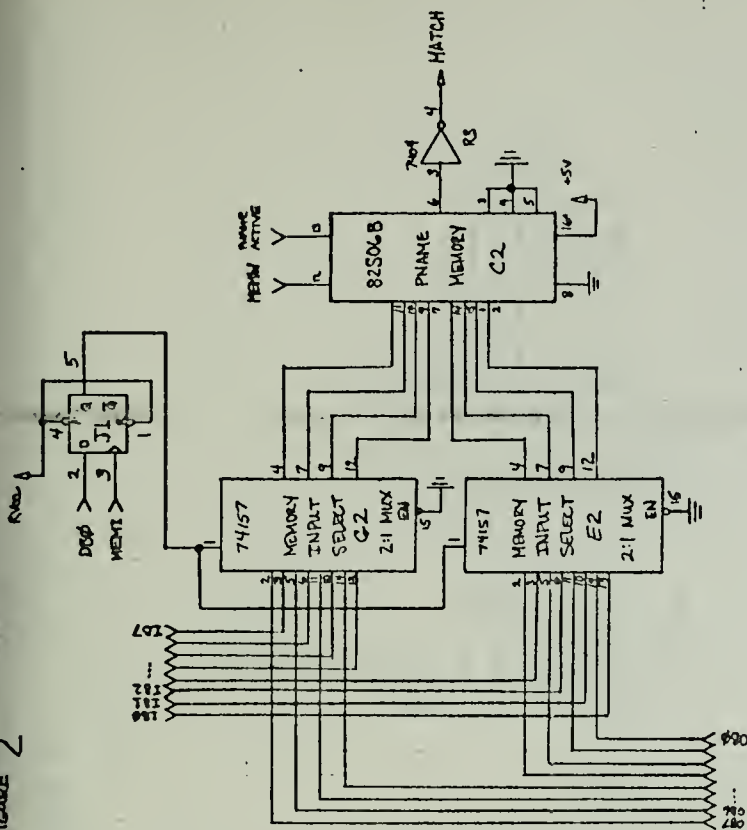
RING INTERFACE FUNCTIONAL OVERVIEW

Michael J. Hanna	22 MAY 76
------------------	-----------

2500021-1



2500021-1



NAVAL	POST GRADUATE	SCHOOL
-------	---------------	--------

3) NODE NUMBER MATCH CIRCUITRY

2) PNAME MEMORY CIRCUITRY

Michael J. Horvitz	21 MAY 74
--------------------	-----------

FIGURE 2

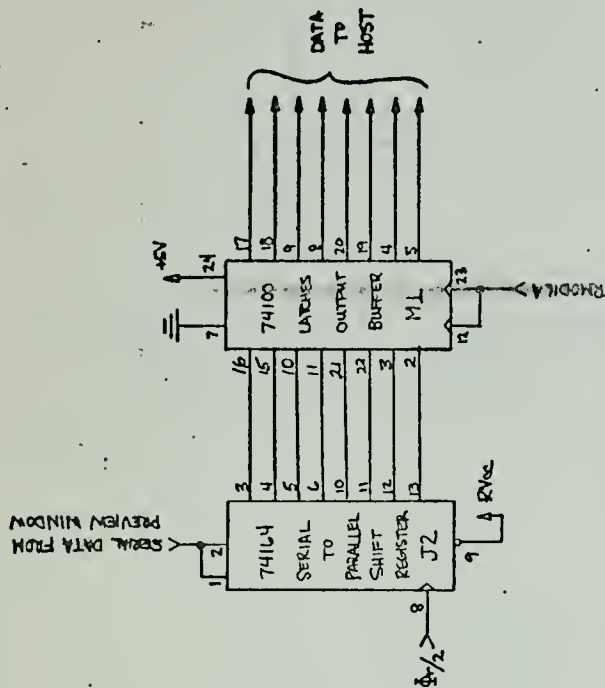


FIGURE 2

1) OUTPUT MULTIPLEXOR COUNTRY

2) OUTPUT BUFFER CIRCUITRY

Michael J. Harris

22 MAY 74

Figure 2

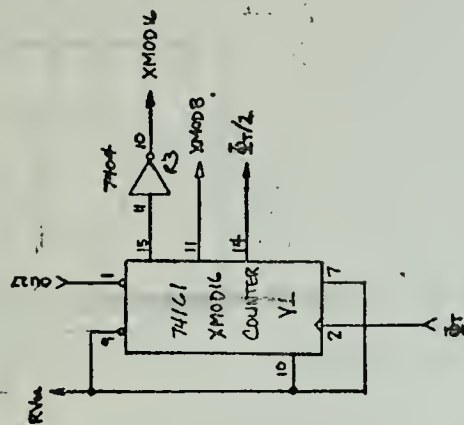


Figure 2

Figure 2

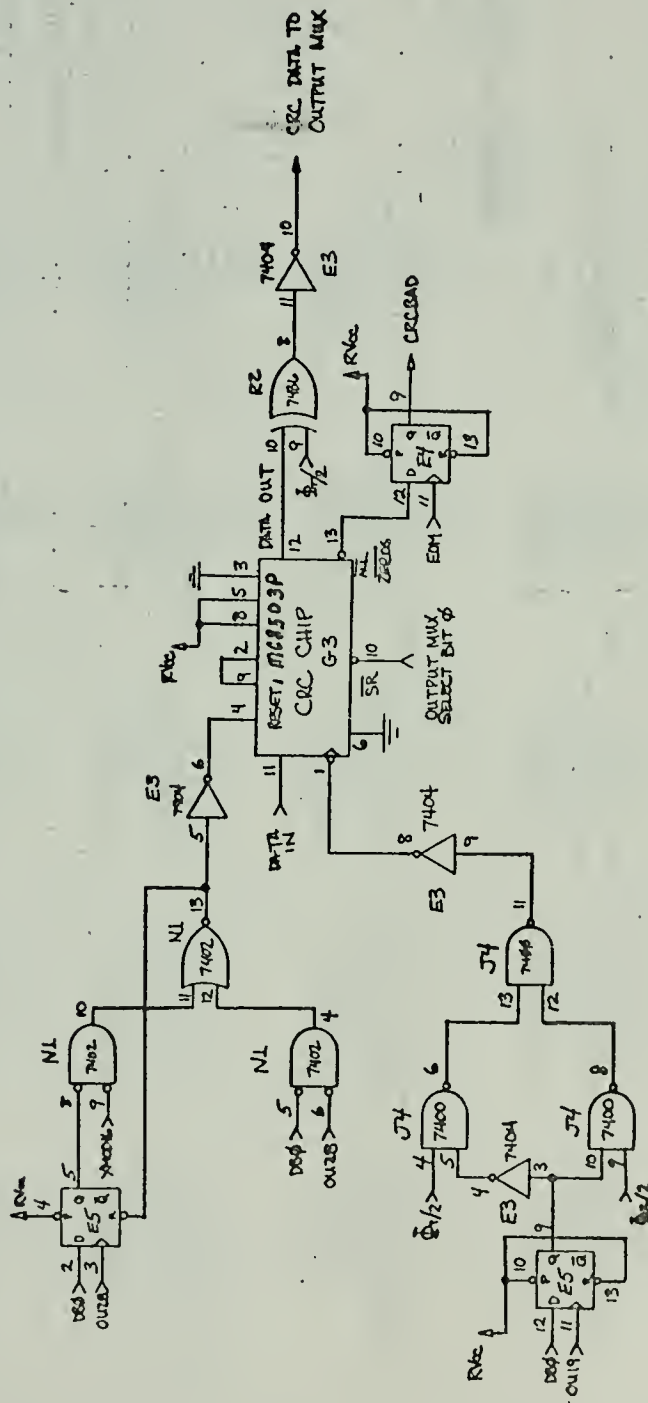


1) THE VALUE OF RA AND C
DETERMINE THE UNIQUE
TIMING PERIOD (PS PER
PAGE 6-52 SKELETICS
MANUAL 1972)

FIGURE 2



1) THE VALUE OF RA AND C
DETERMINE THE UNIQUE
TIMING PERIOD (PS PER
PAGE 6-52 SKELETICS
MANUAL 1972)



NOTES:

- 1) CRCR = 0 IMPUES AN IMMEDIATE CRD RESET
- 2) CECR = 1 IMPUES A CRD RESET ON THE NEXT XMODUL PULSE
- 3) CRC CHIP PIN 7 = GND
PIN 14 = +5V

NAVAL POST GRADUATE SCHOOL

CRC CIRCUITRY

Michael J. Harrier 22 MAY 74

Figure 2

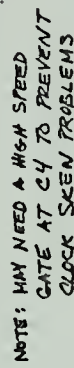
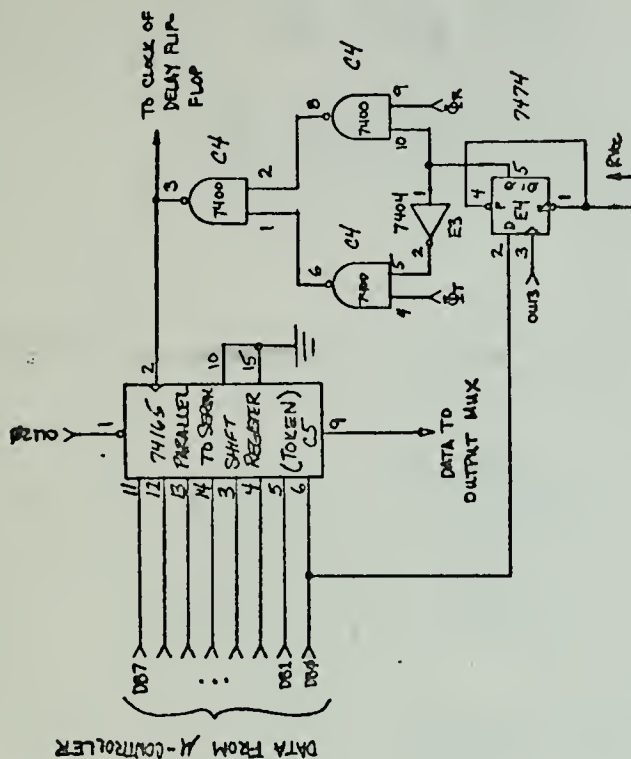


Figure 2

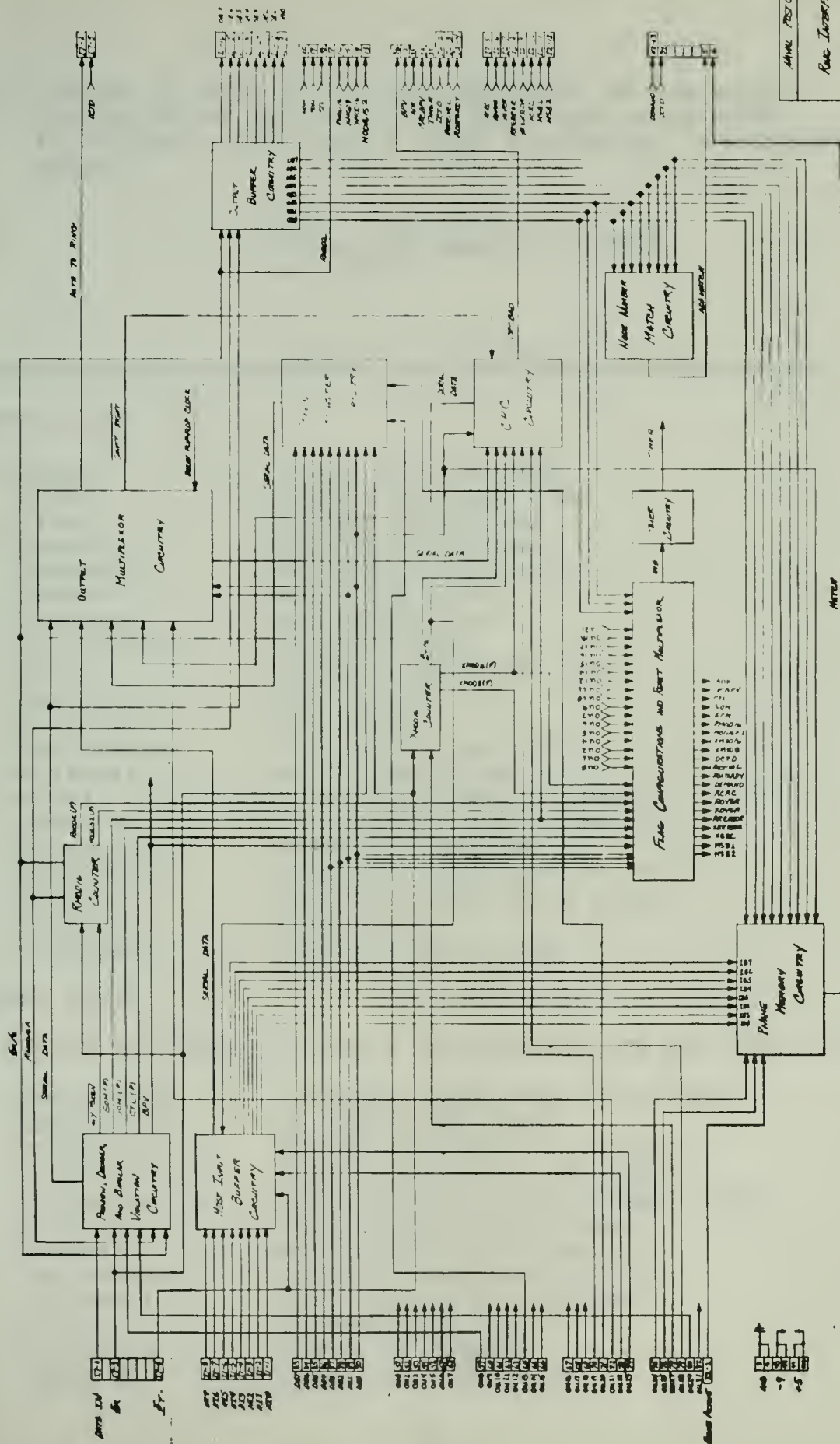


NOTE: MAY NEED A HIGH SPEED GATE AT C4 TO PREVENT CLOCK SKEN PROBLEMS

1) HOST INPUT BUFFER CIRCUITRY

2) TOKEN REGISTER CIRCUITRY

Michael J. Lonie 22 MAY 74



ANALYST CONTINUED SCHOOL
 RUC INTERFERENCE CIRCUITRY
 MANUAL CONVERSIONS
 25 MAY 70


```

0 /THIS PROGRAM IS USED IN THE RING INTERFACE BY THE MICRO-
0 /  CONTROLLER TO HANDLE ALL THE SEQUENCES WHICH HAVE,
0 /  IN PREVIOUS RING SYSTEMS, BEEN IMPLEMENTED IN HARDWARE
0 /  DESIGN.

```

```

0 /MICHAEL J. HARRIS
0 /NAVAL POST GRADUATE SCHOOL
0 /MAY 1974

```

```

0 /DEFINE THE INPUT AND OUTPUT PORTS FOR THE MICRO-CONTROLLER

```

```

0 /INPUT PORTS

```

```

0 EOM=0           /END-OF-MESSAGE FLAG
0 SOM=1           /START-OF-MESSAGE FLAG
0 CTL=2           /CONTROL TOKEN FLAG
0 RM0D2=3         /RECEIVE M0D2 FLAG
0 RM0D16=4        /RECEIVE M0D16 FLAG
0 XM0D3=5         /XMIT M0D3 FLAG
0 XM0D16=6        /XMIT M0D16 FLAG
0 M0D16IS2=7      /M0D16=2 FLAG
0 BPV=8           /BIPOLAR VIOLATION FLAG
0 HACCEPT=9     /HOST ACCEPT LINE
0 XMITL=10        /XMIT LINE
0 HDATARDY=11     /HOST DATA READY LINE
0 ALTER=12        /ALTER PNAME LINE
0 DCT=13          /DISCONNECT-CONNECT LINE
0 MATCH=14        /MATCH PNAME FROM RI MEMORY
0 ADDMATCH=15     /ADDRESS MATCH LINE
0 CRCBAD=16       /CRC ERROR LINE FROM CRC CHIP
0 AOK=17          /RCRC, ROVER, AND BPV=0 LINE
0 TIMER=19        /RI TIMER
0 CRCBPV=20       /RCRC AND BPV=1 LINE

```

```

0 /OUTPUT PORTS

```

```

0 /NOTE THAT PORTS 0 THRU 8 ARE IDENTICAL TO THE INPUT PORTS

```

```

0 RECEIVEL=9      /RECEIVE LINE FROM RI TO HOST
0 RDATARDY=10     /RI DATA READY LINE
0 DEMAND=11       /DEMAND LINE FROM RI TO HOST
0 RCRC=12         /RECEIVE CRC ERROR SET LINE
0 TOKENC=13       /TOKEN REGISTER CLOCK SELECT LINE
0 ROVER=14        /RECEIVE OVERRUN SET LINE
0 XOVER=15        /XMIT OVERRUN SET LINE
0 RRERROR=16      /RECEIVE RING-ERROR SET LINE
0 XRERROR=17      /XMIT RING-ERROR SET LINE
0 DCTD=18         /DISCONNECTED FLAG
0 CRCCLK=19       /CRC CLOCK SELECT
0 TOKEN=20        /TOKEN REGISTER LOAD

```



```

OUTPUT=21          /RI OUTPUT MODE SELECT
HOSTB=22           /HOST BUFFER LOAD LINE
HOSTS=23           /HOST SHIFT REGISTER LOAD LINE
MEMI=24            /MEMORY INPUT SELECT
MEMW=25            /MEMORY WRITE LINE
XMOD16C=27         /XMIT MOD16 COUNTER RESET
CRCR=28            /CRC CHIP RESET
PREVIEW=29         /PREVIEW WINDOW RESET (G0BBLE) LINE
RMUX=31            /RESET MULTIPLEXOR

```

/RESET MULTIPLEXOR PORT DEFINITIONS

```
RTIMER=0          /RI TIMER RESET LINE
XCRC=1            /XMIT CRC ERROR SET LINE
MSB1=2            /MESSAGE STATUS BIT 1 SET LINE
MSB2=3            /MESSAGE STATUS BIT 2 SET LINE
```

/END OF PORT DEFINITIONS

MISCELLANEOUS DEFINITIONS

```

                                /RI NODE NUMBER = 0
RINUM1=0AAH                    /1ST HALF OF NODE NUMBER (LOW ORDER)
RINUM2=0AAH                    /2ND HALF OF NODE NUMBER (HIGH ORDER)

```

/END OF MISCELLANEOUS DEFINITIONS

/INITIALIZATION ROUTINE

```

MAIN,      -DCTD=: M1          /CONTINUE IF CONNECTED
           DCT =: *            /WAIT UNTIL HOST SIGNALS CONNECT
           =: ENTER            /CALL THE ENTER PROCEDURE
M1,        RMUX := RTIMER       /RESET RI TIMER
           TOKENC := 1          /SELECT TOKEN CLOCK = "RECEIVE"
M2,        TIMER=: M3          /
           CTL =: XMIT          / WAIT FOR TIMER,
           SOM =: RECEIVE       /  CTL, SOM,
           DCT =: EXIT          /  OR DISCONNECT
           =: M2
M3,        XM0D16C := 0         /RESET XM0D16 COUNTER
           XM0D3 := 0          /RESET XM0D3 FLAG
           TOKEN := 0EAH       /PUT CTL INTO TOKEN REGISTER
           OUTPUT := 2         /SELECT OUTPUT MODE = "TOKEN"
           -XM0D3 =: *         /WAIT FOR XM0D3 FLAG = 1
           OUTPUT := 3         /SELECT OUTPUT MODE = "RELAY"
           =: MAIN             /GO TO MAIN

```


``` /EXIT PROCEDURE ```

```

#13 EXIT,      RMUX := RTIMER /RESET RI TIMER
#14 EX1,      TIMER := EX2 / WAIT FOR CTL, TIMER, SOM,
#15           SOM := EXIT / OR CONNECT FROM HOST
#16           -DCT := MAIN /
#17           -CTL := EX1 /
#18           PREVIEW := 0 /GOBBLE CTL
#19 EX2,      DCTD := 1 /SET DISCONNECTED FLAG TO 1
#1A           := MAIN /GO TO MAIN

```

``` /ENTER PROCEDURE ```

```

#1B ENTER,    RECEIVED := 0 /RESET FLAGS
#1C           RDATAIDY := 0
#1D           DEMAND := 0
#1E           ROVER := 0
#1F           XOVER := 0
#20           RRERROR := 0
#21           XRERROR := 0
#22           EOM := 0
#23           SOM := 0
#24           CTL := 0
#25 E1,      RMUX := RTIMER /RESET RI TIMER
#26 E2,      CTL := E3 /
#27           TIMER := E4 /WAIT FOR CTL, TIMER,
#28           SOM := E1 /SOM, OR DISCONNECT FROM HOST
#29           DCT := MAIN /
#2A           := E2
#2B E3,      XM0D16C := 0 /RESET XM0D16 COUNTER
#2C           XM0D8 := 0 /RESET XM0D8 FLAG
#2D           CTL := 0 /RESET CLT FLAG
#2E           -XM0D8 := * /WAIT FOR CTL TO PASS
#2F E4,      DCTD := 0 /SET DISCONNECTED FLAG TO 0
#30           := MAIN

```

``` /XMIT PROCEDURE ```

```

#31 XMIT,     XOVER := 0 /RESET FLAGS
#32           XRERROR := 0
#33           DEMAND := 0
#34           BPV := 0
#35           EOM := 0
#36           CTL := 0
#37           RMUX := RTIMER /RESET RI TIMER
#38           TOKENC := 0 /SELECT TOKEN CLOCK = "XMIT"
#39           DCTD := MAIN /GO TO MAIN IF HOST SIGNALS DISCONNECT
#3A           -ALTER := X5 /BEGIN XMITTING IF ALTER=0
#3B
#3B           /ALTER PNAME SEQUENCE
#3B X1,      DEMAND := 1 /RAISE DEMAND LINE
#3C X2,      -ALTER := X3 /WAIT FOR HOST TO DROP ALTER LINE
#3D           -TIMER := X2 /OR FOR RI TIMER TO EXPIRE
#3E

```



```

      =: DIE
BF X3,  HOSTB := 0      /TAKE DATA INTO HOST BUFFER
      DEMAND := 0      /DROP DEMAND LINE
      MEMI := 1        /O=> INPUT FROM CONTRÖLLER, I=> HOST
      MEMW := 0        /PULSE MEMORY WRITE LINE
      =>
      =>
      =>      /GIVE HOST TIME TO RAISE
      =>      /ALTER LINE
      =>
      3
      9 ALTER :=: X1      /IF ALTER=1 THEN GO AGAIN
A X4,  XM0D16C := 0      /RESET XM0D16 COUNTER
      XM0D8 := 0        /RESET XM0D8 FLAG
      OUTPUT := 2       /SELECT OUTPUT = "TOKEN"
      TOKEN := 0EAH     /PUT CTL INTO TOKEN REGISTER
      -XM0D8 :=: *      /WAIT FOR XM0D8 FLAG=1
      OUTPUT := 3       /SELECT OUTPUT = "RELAY"
      0
      1 =: MAIN
      1
      1 /XMIT SEQUENCE
      1
      1 X5,  -XMITL :=: X4      /IF HOST DOESN'T WANT TO XMIT,
      2      /JUST OUTPUT CTL AND GO TO MAIN
      2 XM0D16C := 0      /RESET XM0D 16 COUNTER
      3 XM0D16 := 0      /RESET XM0D16 FLAG
      4 OUTPUT := 2       /SELECT OUTPUT = "TOKEN"
      5 -XM0D16 :=: *      /
      6 XM0D16 := 0      /WAIT FOR DELAY OF 2 XM0D16
      7 -XM0D16 :=: *      /TIME PERIODS
      8 XM0D16 := 0      /
      9 TOKEN := 0ECH     /PUT SOM INTO TOKEN REGISTER
      A CRCR := 1        /RESET CRC CHIP DELAYED
      B CRCCLK := 0      /SELECT XMIT/2 CLOCK FOR CRC CHIP
      C
      C /XMIT HANDSHAKING SEQUENCE
      C
      C X6,  DEMAND := 1      /RAISE DEMAND DATA LINE
      D X7,  XM0D16 :=: XMITERR /OVERRUN
      E      -HDATARDY :=: X7 /WAIT FOR HOST DATA READY=1
      F      HOSTB := 0      /GRAB DATA INTO HOST BUFFER
      0 XM0D16 :=: XMITERR /OVERRUN??
      1 DEMAND := 0      /TELL HOST YOU GOT DATA
      2 -XM0D16 :=: *      /WAIT FOR XM0D16 FLAG
      3 HOSTS := 0      /PUT DATA INTO SHIFT REGISTER
      4 OUTPUT := 1      /SELECT OUTPUT = "HOST"
      5 XM0D16 := 0      /RESET XM0D16 FLAG
      6 X8,  XM0D16 :=: XMITERR /OVERRUN
      7 HDATARDY :=: X3 /WAIT FOR HOST DATA READY=0
      8 -XMITL :=: X9      /MORE TO XMIT?????
      9 -TIMER :=: X6      /IF TIMER STILL OKAY...GO AGAIN
      A =: DIE      /ELSE DIE
      B X9,  -XM0D16 :=: *      /WAIT FOR XM0D16
      C OUTPUT := 0      /SELECT OUTPUT = "CRC"
      D

```



```

6E      XM0D16 := 0      /RESET XM0D16 FLAG
6F      -XM0D16 :=: *    /WAIT FOR CRC TO XMIT TO RING
70      XM0D16 := 0      /
71      -XM0D16 :=: *    /
71
71      /END OF XMIT...EOM WATCH PROCEDURE MUST FOLLOW
71      /IMMEDIATELY IN MEMORY
71
71 /EOMWATCH PROCEDURE
71
71 EOMWATCH,      RMUX := RTIMER /RESET TIMER
72      TOKEN := 0E5H /PUT EOM INTO TOKEN REGISTER
73      OUTPUT := 2 /SELECT OUTPUT = "TOKEN"
74      XM0D16 := 0 /RESET XM0D16 FLAG
75      -XM0D16 :=: * /WAIT FOR XM0D16=1
76      XM0D16 := 0 /RESET XM0D16 FLAG
77      XM0D8 := 0 /RESET XM0D8 FLAG
78      TOKEN := RINUM1 /PUT 1ST HALF OF ADDRESS INTO
79      /TOKEN REGISTER
79      -XM0D8 :=: * /WAIT FOR XM0D8 =1
7A      TOKEN := RINUM2 /PUT 2ND HALF OF ADDRESS INTO
7B      /TOKEN REGISTER
7B      -XM0D16 :=: * /WAIT FOR XM0D16=1
7C      TOKEN := 0EAH /PUT CTL INTO TOKEN REGISTER
7D      EOM :=: EW2 /IF EOM FLAG=1 THEN EW1
7E EW1,      CTL :=: XRINGERR /WAIT FOR EOM,
7F      TIMER:=:XRINGERR /CTL, OR TIMER
80      -EOM :=: EW1
81 EW2,      SOM := 0 /RESET SOM FLAG
82      M0D16IS2 := 0 /RESET M0D16IS2 FLAG
83 EW2A,      CTL :=: XRINGERR /WAIT FOR CTL,
84      TIMER:=:XRINGERR /TIMER, OR M0D16IS2
85      -M0D16IS2:=: EW2A
86      -ADDMATCH:=:XRINGERR /CHECK ADDRESS MATCH
87      RM0D16 := 0 /RESET RM0D16 FLAG
88      -RM0D16 :=: * /WAIT FOR RM0D16=1
89      OUTPUT := 3 /SELECT OUTPUT ="RELAY"
9A      RMUX := MSB1 /
9B      RMUX := MSB2 /SET STATUS BITS FOR HOST
9C      RMUX := XCRC /
9D      BPV :=: XRINGERR / CHECK FOR BIPOLAR VIOLATION
9E
9E      /STATUS SEQUENCE TO HOST
9E      XM0D16C := 0 /RESET XM0D16 COUNTER
9F      XM0D16 := 0 /RESET XM0D16 FLAG
90      DEMAND := 1 /RAISE DEMAND DATA LINE
91 EW3,      XM0D16 :=: EW5 /OVERRUN???
92      -HDATARDY:=:EW3 /WAIT FOR HOST DATA READY=1
93      DEMAND := 0 /DROP DEMAND DATA LINE
94 EW4,      XM0D16 :=: EW5 /OVERRUN???
95      HDATARDY :=: EW4 /WAIT FOR HOST DATA READY=0
96      =: MAIN
97 EW5,      KOVER := 1 /JUST SET OVERRUN FLAG
98      =: MAIN /AND GO TO MAIN
99

```



```

99 /XMIT ERROR PROCEDURE
99
99 XMITERR,      XOVER := 1      /SET XMIT OVERRUN FLAG
9A              TOKEN := OFFH    /PUT ONES INTO TOKEN REGISTER
9B              OUTPUT := 2      /SELECT OUTPUT ="TOKEN"
9C              XMØD16 := 0      /RESET XMØD16 FLAG
9D              -XMØD16 :=: *    /WAIT FØR XMØD16=1
9E              :=: EØMWATCH
9F
9F /XMIT RING ERROR PROCEDURE
9F
9F XRINGERR,     XRERRØR := 1    /SET XMIT RING ERROR FLAG
A0 XR1,          RMUX := RTIMER  /RESET RI TIMER
A1 XR2,          TIMER :=: XR3   /
A2              SØM :=: XR1      /WAIT FØR CTL,SØM
A3              -CTL :=: XR2     /ØR TIMER
A4              CTL := 0         /RESET CTL FLAG
A5              PREVIEW := 0     /GØBBLE CTL
A6 XR3,          OUTPUT := 3     /SELECT OUTPUT ="RELAY"
A7              :=: MAIN
A8
A8 /RECEIVE PROCEDURE
A8
A8 RECEIVE,      RDATARDY := 0    /RESET FLAGS
A9              RCRC := 0
A9              RØVER := 0
AB              RRERRØR := 0
AC              MØD16IS2 := 0
AD              EØM := 0
AE              SØM := 0         /RESET SØM FLAG
AF              BPV := 0
B0              RMUX := RTIMER  /RESET RI TIMER
B1 R1,          CTL :=: RRINGERR /
B2              SØM :=: RRINGERR /WAIT FØR CTL,SØM,
B3              TIMER :=: RRINGERR /TIMER, ØR MØD16IS2
B4              -MØD16IS2 :=: R1 /
B5              RMØD16 := 0      /RESET RMØD16 FLAG
B6              MEM1 := 0        /SELECT MEMORY INPUT FRØM RING
B7              -MATCH :=: MAIN  /IF NØ MATCH, GØ TØ MAIN
B8              RECEIVEL := 1    /RAISE RECEIVE LINE SINCE MESSAGE
B9              /IS FØR US
B9              RDATARDY := 1    /RAISE RECEIVE DATA READY LINE
BA R2,          HACCEPT :=: R3 /
BB              -RMØD16 :=: R2   /WAIT FØR HØST ACCEPT ØR OVERRUN
BC              :=: R4A         /OVERRUN!!!!
BD R3,          RDATARDY := 0    /DRØP RECEIVE DATA READY LINE
BE R4,          -HACCEPT :=: R5 /
BF              -RMØD16 :=: R4   /WAIT FØR HØST ACCEPT=0 ØR OVERRUN
C0 R4A,         RØVER := 1       /OVERRUN!!!
C1              :=: R6
C2 R5,          RMØD16 :=: R4A   /OVERRUN??
C3 R5A,         TIMER :=: RRINGERR
C4              EØM :=: R9       /WAIT FØR RMØD16,TIMER,EØM,
C5

```



```

C6 CTL :=: RRINGERR /CTL, ØR SØM
C7 SØM :=: RRINGERR /
C8 -RMØD16 :=: R5A
C9 CRCCLK := 1 /SELECT RECEIVE/2 CLOCK FØR CRC CHIP
CA R6, CRCR := 0 /RESET CRC CHIP IMMEDIATE
CB RMØD16 := 0 /RESET RMØD16 FLAG
CC R7, RDATA RY := 1 /RAISE RECEIVE DATA READY LINE
CD HACCEPT :=: R7A /
CE -RMØD16 :=: R7 /WAIT FØR HACCEPT=1 ØR ØVERRUN
CF R7A, :=: R4A /ØVERRUN!!!!
D0 R8, RDATA RY := 0 /DRØP RECEIVE DATA READY LINE
D1 -HACCEPT :=: R5 /WAIT FØR HACCEPT=0 ØR
D2 -RMØD16 :=: R8 /ØVERRUN???
D3 R9, :=: R4A /ØVERRUN!!!!
D4 TØKENC := 1 /SELECT TØKEN CLOCK = "RECEIVE"
D5 -CRCBAD :=: R10 /CHECK FØR CRC ERROR
D6 R10, RCRC := 1 /CRC ERROR IN RECEIVED MESSAGE!!
D7 TIMER :=: RRINGERR /WAIT FØR TIMER
D8 -RMØD16 :=: R10 /ØR RMØD16
D9 RMØD16 := 0 /WAIT FØR NØDE NUMBER
DA -RMØD16 :=: * /TØ PASS
DB TØKEN := 55H /PUT ØNES INTO TØKEN REGISTER
DC -AØK :=: R11 /DID WE GET MESSAGE PRØPERLY??
DD ØUTPUT := 2 /SELECT ØUTPUT = "TØKEN"
DE -RMØD2 :=: * /WAIT FØR 2 BITS TØ PASS
DF RMØD2 :=: *
E0 R11, :=: R12
E1 -RMØD2 :=: * /WAIT FØR 2 BITS TØ PASS
E2 RMØD2 :=: *
E3 ØUTPUT := 2 /SELECT ØUTPUT="TØKEN"
E4 -RMØD2 :=: * /WAIT FØR ØNE TØ SHIFT TØ RING
E5 RMØD2 :=: *
E6 -CRCBPV :=: R12 /TEST FØR CRC AND BPV
E7 -RMØD2 :=: * /WAIT FØR THE LAST ØNE TØ SHIFT ØUT
E8 R12, ØUTPUT := 3 /SELECT ØUTPUT="RELAY" MØDE
E9 R13, RECEIVEL := 0 /DRØP RECEIVE LINE
FA RDATA RY := 1 /
FB -HACCEPT :=: * /PASS STATUS INFØRMATION
FC RDATA RY := 0 /TØ HØST
FD HACCEPT :=: * /
FE :=: MAIN
FF /RECEIVE RING ERROR PRØCEDURE
F R RRINGERR, RRERROR := 1 /SET RECEIVE RING ERROR FLAG
F0 RECEIVEL := 0 /DRØP RECEIVE LINE
F1 RR1, RMUX := RTIMER /RESET RI TIMER
F2 RR2, SØM :=: RR1 /
F3 TIMER :=: MAIN /WAIT FØR CTL,SØM , ØR TIMER
F4 -CTL :=: RR2 /
F5 CTL := 0 /RESET CTL FLAG
F6 PREVIEU := 0 /GØBBLE CTL TØKEN
F7 :=: MAIN
F8

```



```

F8 /DIE ROUTINE
F8
F8 DIE,      OUTPUT := 3      /SELECT OUTPUT = "RELAY" MODE
F9 D1,      RMUX := RTIMER    /RESET THE RI TIMER
FA D2,      TIMER := D3      /
FB          SØM := D1        /WAIT FOR TIMER, SØM, ØR CTL
FC          -CTL := D2      /
FD          PREVIEW := 0     /GOBBLE CTL TOKEN
FE D3,      DCTD := 1        /SET DISCONNECTED FLAG TØ 1
FF          := *            /WAIT FOR RESET
100
100
100
100 /END OF PROGRAM.....
100 SS

```


HEX FORMATTED HIGH ORDER BYTE

```

00000000D0ACDB29F0012AC3DBEB29F041A0B0AFE
0001000DA0A9F00ACBED2DD020D9F161514111036
00020000F0E1F1E1D00BDACBEB29F041A1DDA0DBF
00030009F100E14171F1D0012ADD314D3CC9F09AF
00040001407069F9F9F9F9F9FB3041A0A0BDA0A0B
00050009FD504190AD919D9190B030C14B9D4095D
0006000B914D9030A19B9B4D5CC9FD90A19D91924
0007000D9000B0A19D9191A0BDA0BD90BBFBDAC71
0008000DF1E18BDACD3D01BDB0A000000B7041976
000900014B9D414B9B49F109F100B0A19D99F0E2C
000A00000ACBEDD1D020A9F1513110F181F1E173D
000B00000BDBEACD31B07D11615B6DB9F15D6DB2D
000C000119FBBACBF3DBEDB0C031B15B6DB9F1530
000D000D6DB9F12CF13ACDB1BDB0BCE0ADCBC9F45
000E000DCBC0ADCBCCBDCBC0A1615D615B69F0FEF
000F0001600BEACDD1D029F0A00ACBEDD020D9FE6

```

HEX FORMATTED LOW ORDER BYTE

```

0000000FEFCFAFCE4FFFFEF3CE57ECF8FFFF15FD13
0001000EFFCFDFFE6ECFDEBFFFEFDFFFFFFFFFFFF4A
0002000FFFFFFFFFFFFFD4D0DAFDD9FFFFFFFFD1FFB5
0003000FDFFFFFFFFFFFFFFFFFFFFDAEFEC0C307FF99
0004000FFFEFFB5BAB9B8B7B6C4FFFFFD15B1FCE0
0005000FDB5FFFFFDAFFFA8FF13FEFFFE66A2FF8E
000600066FF9DFFFEFF669994A30794FFFF91FF33
00070003FFF1AFDFF8AFFFF55365584157E60604D
000800031FFFF60607C60FF77FCFDFCFE60FFFF8E
0009000FE636EFF636BFDFFDFDFF00FDFF628EFEDA
000A000FF595F5EFFFFFCFDFFFFFFFFFFFFFFFFFFFF4C
000B000FF1010104EFFFFFDFFFE42453FFF3D4189
000C000FE353F102C10103CFEFFFFFE30333FFF8B
000D0003D2F3FFE29FE1029FF26AA1FFD222117D2
000E0001F1EFD1C1B171918FCFFFE14FF12FDFF3E
000F000FFFF0EFD0DFFFFFDFCFF010605FFFE00EB

```


BNPF MACHINE CODE FOR HIGH ORDER BYTE

4096	BNNNNPPNPF	BNNNNPPNPF	BPPNPPNPF	BPNNPPNPF
4100	BPNNPPPPPF	BNNNNNNNNF	BNNNPPNPF	BPNNPPNPF
4104	BPNNPPPPPF	BPNNPPPPPF	BPNNPPNPF	BPNNPPPPPF
4108	BNNNNPPNPF	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF
4112	BPNNPPNPF	BNNNPPNPF	BPNNPPPPPF	BNNNNNNNNF
4116	BPNNPPNPF	BPNNPPPPPF	BPNNPPNPF	BPNNPPPPPF
4120	BNNNNNNPF	BNNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4124	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF
4128	BNNNPPPPPF	BNNNPPPPPF	BNNNPPPPPF	BNNNPPPPPF
4132	BNNNPPNPF	BNNNNNNNNF	BPNNPPNPF	BPNNPPNPF
4136	BPNNPPPPPF	BPNNPPNPF	BPNNPPPPPF	BNNNNNNPF
4140	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF
4144	BPNNPPPPPF	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF
4148	BNNNPPPPPF	BNNNPPPPPF	BNNNPPNPF	BNNNNNNNNF
4152	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF	BNNNPPNPF
4156	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4160	BNNNPPNPF	BNNNNNNNNF	BNNNNNNPF	BPNNPPPPPF
4164	BPNNPPPPPF	BPNNPPPPPF	BPNNPPPPPF	BPNNPPPPPF
4168	BPNNPPPPPF	BPNNPPNPF	BNNNNNNPF	BNNNPPNPF
4172	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF
4176	BPNNPPPPPF	BPNNPPNPF	BNNNNNNPF	BNNNPPNPF
4180	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF	BPNNPPNPF
4184	BNNNPPNPF	BNNNPPNPF	BNNNNNNPF	BNNNPPNPF
4188	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF	BNNNPPNPF
4192	BPNNPPNPF	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF
4196	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF
4200	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BPNNPPNPF
4204	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF
4208	BPNNPPNPF	BNNNNNNNNF	BNNNPPNPF	BNNNPPNPF
4212	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF	BNNNPPNPF
4216	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF	BPNNPPNPF
4220	BNNNPPNPF	BPNNPPPPPF	BPNNPPNPF	BPNNPPNPF
4224	BPNNPPPPPF	BNNNPPPPPF	BNNNPPNPF	BPNNPPNPF
4228	BPNNPPNPF	BPNNPPNPF	BPNNPPNPF	BNNNPPNPF
4232	BPNNPPNPF	BNNNPPNPF	BNNNNNNNNF	BNNNNNNNNF
4236	BNNNNNNNNF	BPNNPPPPPF	BNNNNNNPF	BNNNPPNPF
4240	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF	BNNNPPNPF
4244	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4248	BPNNPPPPPF	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF
4252	BNNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4256	BNNNNNNNNF	BPNNPPNPF	BPNNPPPPPF	BPNNPPNPF
4260	BNNNPPNPF	BNNNNNNPF	BNNNPPNPF	BPNNPPPPPF
4264	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF	BNNNPPPPPF
4268	BNNNPPNPF	BNNNPPPPPF	BNNNPPNPF	BNNNPPPPPF
4272	BNNNNNNNNF	BPNNPPNPF	BPNNPPPPPF	BPNNPPNPF
4276	BPNNPPNPF	BNNNPPNPF	BNNNNNNPF	BPNNPPNPF
4280	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF
4284	BPNNPPPPPF	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF
4288	BNNNPPNPF	BPNNPPPPPF	BPNNPPNPF	BPNNPPNPF
4292	BPNNPPPPPF	BPNNPPNPF	BPNNPPNPF	BPNNPPNPF
4296	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF
4300	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4304	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPNPF
4308	BPNNPPPPPF	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF
4312	BNNNPPNPF	BPNNPPNPF	BNNNPPNPF	BPNNPPNPF
4316	BNNNPPNPF	BPNNPPNPF	BPNNPPNPF	BPNNPPPPPF
4320	BPNNPPNPF	BPNNPPNPF	BNNNPPNPF	BPNNPPNPF
4324	BPNNPPNPF	BPNNPPNPF	BPNNPPNPF	BPNNPPNPF
4328	BNNNPPNPF	BNNNPPNPF	BNNNPPNPF	BPNNPPNPF
4332	BNNNPPNPF	BPNNPPNPF	BPNNPPPPPF	BNNNPPPPPF
4336	BNNNPPNPF	BNNNNNNNNF	BPNNPPNPF	BPNNPPNPF
4340	BPNNPPNPF	BNNNPPNPF	BNNNNNNPF	BPNNPPPPPF
4344	BNNNPPNPF	BNNNNNNNNF	BPNNPPNPF	BPNNPPPPPF
4348	BPNNPPNPF	BNNNNNNPF	BNNNPPNPF	BPNNPPPPPF

BMPF MACHINE CODE FOR LOW ORDER BYTE

[illegible]

BIBLIOGRAPHY

1. Hirt, K.A., A Prototype Ring-Structured Computer Network Using Micro-Computers, Master's Thesis, Naval Postgraduate School, Monterey, 1973.
2. Loomis, D.C., Ring Communications Protocols, UC Irvine Distributed Computer Project, Memo 46-A, May 1972.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3. Chairman of Computer Science Academic Computing Center U. S. Naval Academy Annapolis, Maryland 21402	1
4. Asst. Professor Ray Brubaker (Code 72Bh) Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5. ENS Gary M. Raetz, USN (Code 72Rr) Computer Science Group Naval Postgraduate School Monterey, California 93940	1
6. ENS Michael J. Harris, USN 3750 S. Maple Grove Rd. Rt #3 Boise, Idaho 83705	1
7. Chairman, Computer Science Group Code 72 Naval Postgraduate School Monterey, California 93940	1



NO 75
26 JUL 76

23800
23748

152417

Thesis
H29126 Harris
c.1

A prototype ring in-
terface for the NPS data
communication ring.

NO 75
26 JUL 76

23800
23748

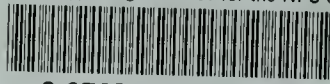
Thesis
H29126 Harris
c.1

A prototype ring in-
terface for the NPS data
communication ring.

152417

thesH29126

A prototype ring interface for the NPS d



3 2768 002 08234 9

DUDLEY KNOX LIBRARY